Enforcing Role-Based and Category-Based Access Control In Java
A Hybrid Approach

Ali, Asad

Awarding institution:
King's College London

KING'S COLLEGE LONDON

Enforcing Role-Based and Category-Based Access Control In Java: A Hybrid Approach

by

Asad Ali

Supervised by Professor Maribel Fernández

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Department of Informatics

in the

Faculty of Natural & Mathematical Sciences

September 2014

ABSTRACT

Access control policies often are partly static, i.e. no dependence on any run-time information, and partly dynamic. However, they are usually enforced dynamically - even the static parts. We propose a new hybrid approach to policy enforcement in the Category-Based Access Control (CBAC) meta-model. We first tackle the challenge of static enforcement of policies following the Role-Based Access Control (RBAC) model, then build on this to enforce, using a hybrid approach, policies following the CBAC model. For the former case, the static approach we advocate includes a new design methodology, for applications involving RBAC, which integrates the security requirements into the system's architecture, helping to ensure that policies are correctly defined and enforced. We apply this new approach to policies restricting calls to methods in Java applications. However, our approach is more general and can be applied to other Object-Oriented languages. We present a language to express RBAC policies on calls to methods in Java, a set of design patterns which Java programs must adhere to for the policy to be enforced statically, and a high-level algorithm for static enforcement. We then adapt and extend this system for hybrid enforcement of CBAC. We modify the static system's policy language, JPol, to specify static and dynamic categories. We establish an equivalence between static categories and static roles (in RBAC), therefore we are able to use the previous design patterns and static verification algorithm, with some adaptations and changes, to enforce static categories. For dynamic categories, we propose a new design methodology and generate code in the target program to do the necessary run-time checks.

Acknowledgements

I would like to sincerely thank my supervisor. She saw potential in me, and her incredibly caring and supportive attitude gave me the ideal environment to reach my potential. Also, without her hard work, often even over weekends, this work would never have been completed. I would also like to thank my friends and family for their support and their sacrifices. They always understood when I could not be there for them, whilst never letting their support for me slip. Last, but not least, I would like to thank my special friend from the depths of my heart. I could never have reached this stage without the invaluable support they have given me every single day for over six years. Through them, I dared to dream and they gave me the confidence and belief to achieve my dreams. If I ever fell, they were always there to pick me up and put me back on the path to success. None of this would have been possible without them.

Contents

List of Figures

Chapter 1

Introduction

The objectives of an access control system are often described in terms of protecting system resources against inappropriate or undesired user access. When there is a request for a resource, the system must check who triggered the request *(authentication)*, check if that user has the permission for the request to be fulfilled (*authorisation*) and as a result allow or deny the request *(enforcement)*. Thus, an implementation of access control requires a specification of the rights associated to users in relation to resources *(a policy)*. The syntax of a policy is specified by a *policy language*, and the structure of a policy is specified by an *access control model*. A wide variety of access control models and policy languages are in use today. Barker's meta-model [5], which we refer to as Category-Based Access Control (CBAC), contains a small amount of abstract primitives, which can be specialised to produce equivalent forms of other models, such as the popular Role-Based Access Control (RBAC) model [21], where each user has one or more roles, and each role has an associated list of permissions on resources.

Our focus is on enforcement, for which there exist two main approaches, static and

dynamic, with a recently emerged third approach combining the two: the hybrid approach. The static approach performs all access checks at compile-time, whereas the dynamic approach performs these at run-time. The traditional approach for enforcing access control is dynamically via a *reference monitor*. This is an application which monitors the execution of a target application, intercepting all the access requests made at run-time. These are then evaluated with the policy to decide if the request is allowed to be executed. If not, the request is not executed and the target program is rolled-back to a safe state.

The evaluation of an access request relies on two associated notions: *session* and *security context*. A session is a self-contained period of execution time in which a single associated and authenticated user can perform user-tasks. Each session records and maintains data about the events that occur in the session e.g. the username of the authenticated user. A security context is session-scoped data associated to a single session, in which security-relevant data is maintained. Dynamic access request evaluation relies on checking the data from the security context held in a session, with the policy. For example, a policy may state that only users with the username *bob* may access resource r, so if there is a request for r, the reference monitor will extract the username from the security context held in the session and check if its value is *bob*. If true, the access is allowed and executed, otherwise it is denied and not executed.

In short, the static approach enables policy violations to be detected earlier, facilitating debugging and reducing the impact on testing, and usually involves a lower run-time cost. However, the kinds of policies enforceable statically are not as expressive nor as flexible as those enforceable by the dynamic approach. Indeed, if the policy contains dynamic attributes, whose values change during execution, it is not possible to enforce those parts of this policy that utilise such attributes at compile-time exclusively. We refer to [28] for a more detailed comparison. The hybrid approach aims to perform as

many access checks as possible at compile-time, which is all the static parts of the policy, followed by all the remaining checks at run-time. Hybrid techniques have recently received attention, due to their ability to leverage the benefits of both approaches (see [11] for an example).

The enforcement mechanism is a crucial part of any application. A small error in the mechanism could expose critical resources to incorrect or even malicious users. As such, the need for a sound methodology for the design and development of policy enforcement mechanisms is well-known in the literature (one example which addresses this issue, in particular, is the model-driven security approach demonstrated in [6]).

The need for dynamic policies, which allow permissions to be specified using information which can change at run-time, is also well-known in the literature. In practice we find that these kinds of policies are commonly partly static and partly dynamic. However, to the best of our knowledge, they are enforced dynamically in current implementations - even the static parts (see, for example, [6, 26, 18]).

Catching errors at an early stage statically aids debugging, reduces testing time and can reduce impact on run-time resources when compared to catching errors only at run-time. Combining this with run-time enforcement will allow the policy to be more expressive; dynamic information, primarily used in permissions, can be expressed and enforced also.

High-Level Problem This thesis addresses the problem of hybrid enforcement of RBAC and CBAC policies in class-based Object-Oriented (OO) programs via the definition of a policy specification language, a design methodology for target programs (in which the policy is to be enforced) to follow, complemented by a combined static and dynamic enforcement mechanism.

1.1 Context

In class-based OO languages, programs are collections of objects that are classified using the notion of a class. An object contains data (fields) and operations (methods). A good practice in OO programming is to ensure interactions with data are through the methods of the class in which the data is declared; data is always access through method calls. This practice helps to achieve the principles of encapsulation (the packing of data and functions into a single component) and information hiding (restricting access to some of the object's components [37]). Many OO languages (e.g. Java, C++) include built-in access control mechanisms for method calls known as *method visibility*. The most common of these mechanisms include assigning a method *private* or *public* visibility; the former meaning it can only be called in the declaring class and thus not in external classes, whilst the latter meaning it can be called in any class. In-built method visibility access control is coarse-grained, therefore there is a need for method-granularity access control (although we are not the first to address this, as discussed in Part I).

In this context, an access control policy restricts calls to methods in classes by users. In the case of an OO application for an organisation, initially the context in which a policy will be specified will consist of the real-world resources and restricting the interactions of users with those resources. This policy will, most often, need to be adapted to the context of the OO application. This gap is usually bridged by implementing the resources as classes and the interactions of users with resources as method invocations on those classes. To refer to the calling user, we will use the terminology *owner*, for cases addressing other kinds of subjects e.g. processes. Therefore an example access to a resource is: owner 'o' calls from a class 'Printer' a method 'print()'.

In this case, as displayed in Figure 1.1, an access control check (taking place either

at compile-time or run-time), of a call to a method in a resource class, will need to establish the following:

1. who is the owner of method call and thus is the subject seeking access to the resource,
2. which of their access rights are active at the time of the request.
3. is the permission to access the resource (i.e. call the method in the resource class) part of the owner's active access rights.

We now consider establishing these parts, firstly dynamically and then statically, in order to investigate how to combine these. The solution for achieving the first two parts at run-time is the previously described reference monitor. When there is a method call to a method in a resource class, the reference monitor establishes the owner by looking up the username and activated access rights (i.e. role in RBAC) in the security context. Then, the policy is consulted to check if the owner has the permission to access the resource, achieving part 3. Although the need for dynamic aspects in policies, requiring run-time enforcement, is well-known, policies commonly contain static aspects also. In this dynamic approach, the static aspects would be enforced at run-time, which may not be needed.

Establishing the first two parts at compile-time, however, is much more difficult. This is because at compile-time, the program has not begun execution therefore there is no concept of a session, no subjects have logged in, no access rights have been activated and there is no run-time security context. The owner of a method call must be known only by data available at compile-time - the program code. However, the users and roles are not usually integrated into the program at the design and source level; they exist only at run-time as part of the (dynamic) security context.

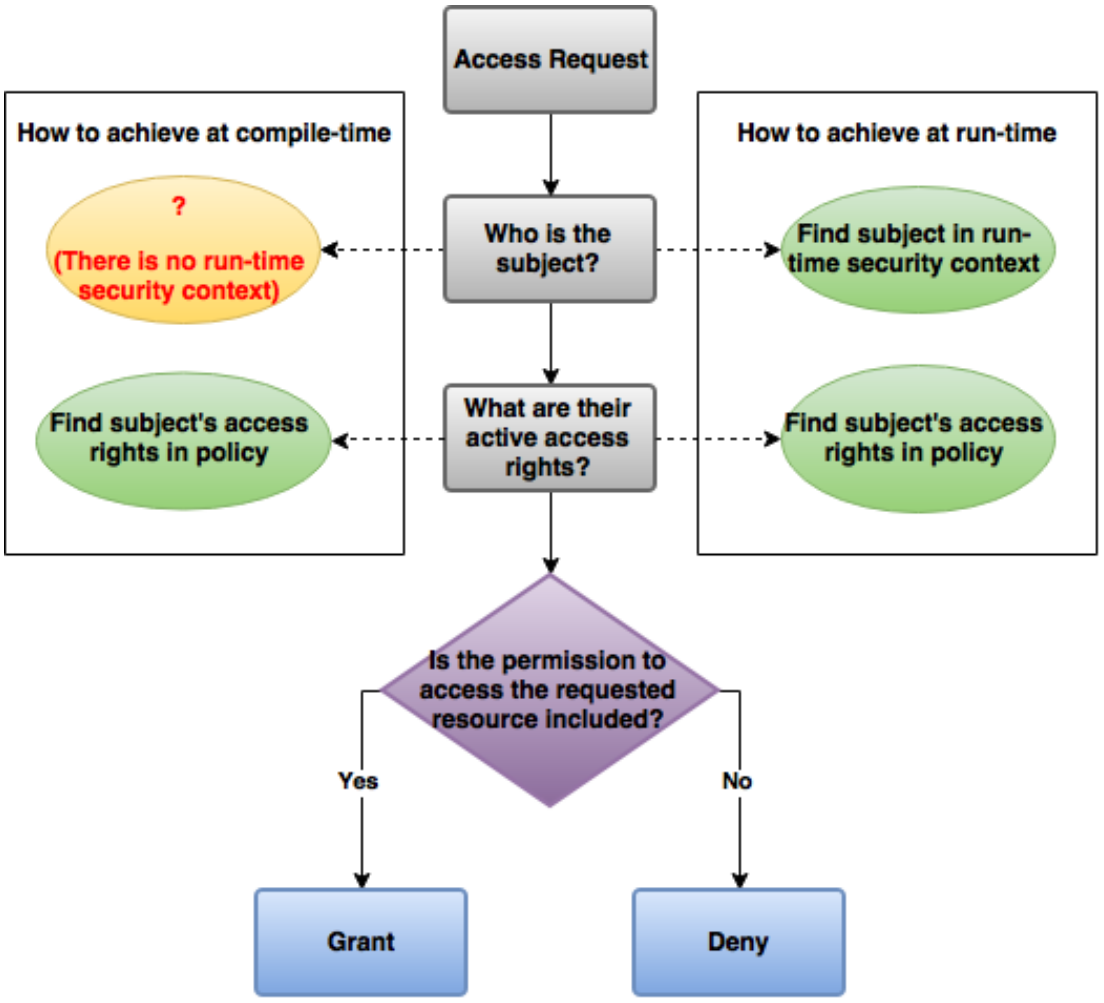

FIGURE 1.1: Comparing static and dynamic (simplified) access request evaluation.

Note that in our approach, we focus on enforcing access control on method calls by roles (or categories in CBAC); the owners/subjects effectively become the roles/categories and we omit the role-to-user (and category-to-user) assignments. This will still protect the resources, since permissions are assigned to roles/categories rather than to users in RBAC/CBAC. As a result, flexibility is allowed for the role/category-to-user assignments to change dynamically, which happens more frequently in practice than changes to resources and role/category permissions.

1.2 Problem

The problem being addressed is establishing the owner of method calls to resource classes, particularly the role that is the owner, to compute at compile-time if method call is allowed by an RBAC policy. This is a major barrier in the static enforcement of RBAC policies. This barrier results in a set of access requests, similar to the one shown in our motivating example below and found in most programs today, being evaluated at run-time when they may not need to be since they require *static permissions*; those that are assigned to roles independently of the state of the system. Being able to establish the owner at compile-time will help to solve the larger problem of enforcing static RBAC policies exclusively at compile-time. This solution would then be applicable to the solving the problem of hybrid enforcement of dynamic CBAC policies. In particular, we consider those that contain some static parts, which is common in practice. The static enforcement mechanism can enforce the static parts, but this can be extended with a dynamic mechanism for enforcing only the dynamic parts of a CBAC policy.

1.2.1 Motivating Example

In order to explain our approach, we introduce a small informal running example of a GP/doctor's surgery where the data is stored according to the Entity-Relationship Model in Figure 1.2. In this example, using RBAC, there are roles 'Admin', 'NHSDoctor' and 'PrivateDoctor'. The Admin should be allowed to read, write and delete NHS and private patient records, appointment records and staff records. The NHSDoctor should be able to read, write and delete NHS visit records (check-ups for NHS patients) and NHS prescriptions, and read NHS patient records. The Private Doctor should be allowed similar operations to the NHSDoctor but for Private patients instead of NHS patients.

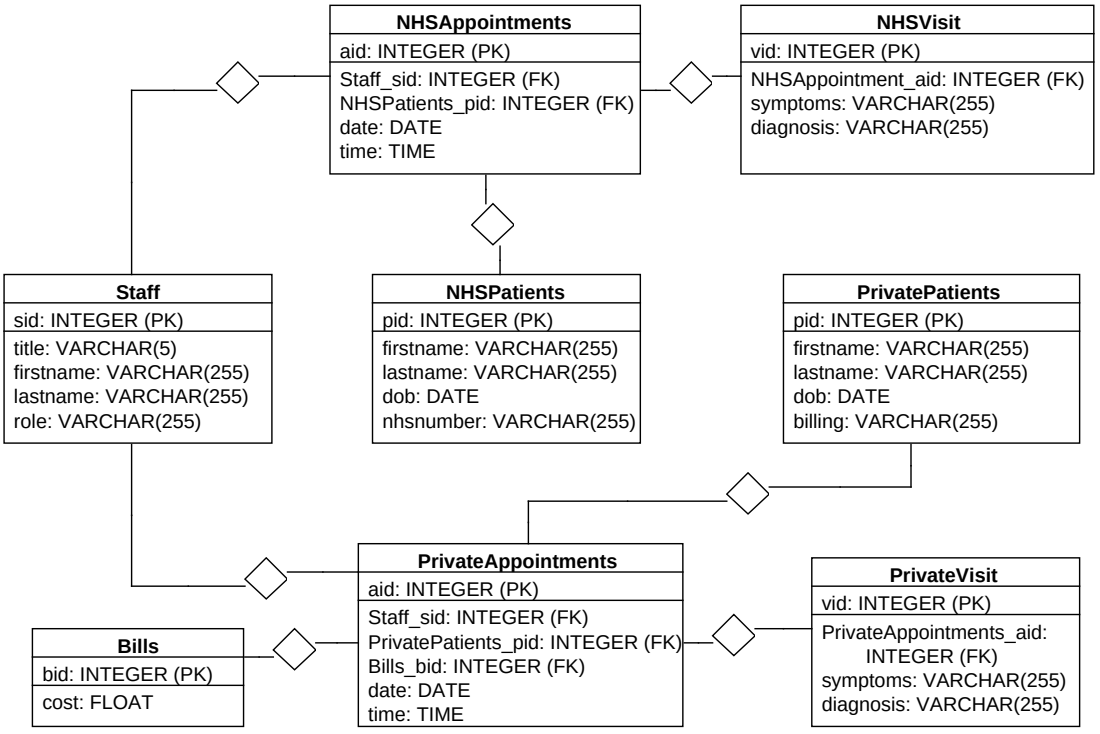

FIGURE 1.2: Entity Relationship Model of GP Surgery Database

To highlight the problem, consider the following Java code:

```
if(securityContext.isUserActiveRole(``NHSDoctor"))
        nhsPatientRecords.getAll();
```

These kinds of code snippets are common in RBAC implementions. In such cases, a programmer would want to be sure that only the authorised role ('NHSDoctor', in this example) can invoke the security-critical, or *protected* method ('nhsPatientRecords.getAll', in this example). This kind of permission is clearly static, because the permission to access the resource does not require any conditions utilising dynamically changing information, but is usually enforced using a dynamic check - the *if* statement - before any such method invocation. The program would then have to be rigorously tested to ensure that for each invocation of a protected method, if a user can reach it at run-time, then the role activated for that user is permitted to perform that protected

invocation in the policy. It would be reasonable to assume that the number of test cases needed would increase as the number of roles increases and the number of invocations of protected methods in the program increases.

Just because this kind of permission is static does not mean it is trivial to statically perform this check. The same is true for statically enforcing a static policy, or the static parts of a partly static policy. Let us start by removing the dynamic check in our example code - the *if* statement - leaving just the invocation of the protected method. We now need to know statically which roles/categories can perform the invocation. The difficulty is that the active role/category exists only in the run-time security context of the program's execution. If we can design and implement the program such that only those roles which are permitted to perform the access requests (requiring static permissions) can actually reach those access request (e.g. only 'NHSDoctor' can reach the invocation 'nhsPatientRecords.getAll'), then these kinds of 'if' statements would not be needed since they would always evaluate to true. To do this, then, we would need to know which regions of the code are accessible by each role/category. However, the roles/categories are rarely integrated into the design and implementation of the program in such a way as to facilitate this.

All previous examples of utilising design-level security techniques in the enforcement of access control policies, which we have examined, rely on completely dynamic enforcement [6] or assume that the owner of a method call is already established (e.g. [39]))

1.3 Hypothesis

We hypothesise that it is possible to statically enforce static (i.e. flat or hierarchical) RBAC policies and static categories in a CBAC policy. More specifically, we hypothesise

that there is a way to design, implement and analyse programs such that it can be statically determined which role/category will be active when an access request (i.e. invocation of a method) is made at run-time. This information can then be utilised to evaluate access requests requiring static permissions via a compile-time algorithm, eliminating the need for conditional statements (e.g. 'if') which check if the active role is the one permitted to access the resource. This static solution can then be extended to evaluate access requests requiring dynamic permissions by inserting code into the program (i.e. 'if' statements with conditions utilising dynamic information).

1.4 Methodology

The methodology which we have follows is discussed as follows. The role/category that is the owner of a call to a method in a resource class can be established through integrating the roles/categories into the program in combination with a static enforcement mechanism. We develop new design patterns which aim to create a flow of the program where all access to methods in resources only occurs in a set of classes specific to each role/category. These classes will constitute role/category-specific interfaces to the program; each role/category will have a single interface through which the user can perform the tasks associated to that role/category, which may invoke protected methods in resource classes. The session is also implemented via a set of classes which constitute a single session interface, through which the user can perform session-related tasks such as logging out. The patterns also group each aspect of the program via naming restrictions. These groups include classes for the: roles/categories, resources, session, security context, runtime monitor and other classes (which do not fit into the other groups). The latter group implement tasks which are not related to the policy or the session - referred to as *other* tasks. Each group will be subject to its own specific set of compile-time checks. Then, at compile-time, when a method call is detected, it

can be computed which class the method belongs to and then which group the class belongs to. This is done via a set of restrictions including naming restrictions and other restrictions to help ensure that at compile-time it is possible to know which class an object belongs to. Importantly, if the computed group is the role/category group, the naming restrictions are able to identify which role/category the class belongs to thus establishing the owning role/category of the call. The validity of the method call will be assessed utilising the group-specific checks.

We conjecture that the dynamic approach is initially less challenging to implement, because it is possible for all access checks to be performed at run-time. However, this is not the case for the static approach, where the requirement is to ensure, at compile time, that no unauthorised access is executed (without performing access checks at run time). Therefore, we first focus on addressing the problem of statically enforcing a completely static policy, following the hierarchical-RBAC model. Since a hierarchical-RBAC (and also flat-RBAC) [22] policy is static, it will contain no dynamic information therefore a program implementing this kind of policy should be able to be checked at compile-time for policy compliance. By extension for the second system, static categories should also be able to be checked at compile-time (and dynamic ones at run-time). Although we have not seen an example of an RBAC policy being enforced using static checks exclusively, this is theoretically possible and indeed we demonstrate it by designing and implementing a static checker for static RBAC policies. Then, we extend this first system by enforcing, using a hybrid approach, a CBAC policy capable of utilising dynamic information (instead of static RBAC). Our hybrid enforcement utilises language-based techniques, namely static analysis, to perform access checks at compile-time, and program transformation, to generate code which will perform remaining checks at run-time.

In both systems, we provide a policy specification language which supports resource

and permission definitions. In the first system, the language also supports role definitions, and also role hierarchies. In the second, the additions are category definitions, category hierarchies and a non-hierarchical relation between categories, *can-be*, which relies on some dynamic condition e.g. an emergency event occurring. To the best of our knowledge, these kinds of policies are typically enforced dynamically in today's available systems (e.g. RBAC in Java Web Security amongst others [6, 26]).

We also develop, *design patterns* for the target program to follow. A design pattern (also referred to as just a *pattern*) serves as a general solution to a specific recurring problem that arises in the design of an application. It provides a guide for the implementation of a solution to a specific problem. A well-known example of a pattern is the Model-View-Controller MVC) pattern [33], which was one of the first patterns to separate user interaction from business logic in applications. In this pattern, there exist three kinds of components (i.e. classes): model, view and controller. The models implement the business logic, the views implement the user-interface and the controllers mediate between the other two. Controllers process events from the views, selecting which model(s) to invoke, interpret the result(s) returned by the model(s) and then selecting the appropriate view(s) to present the result(s) to the user.

1.5 Contributions

In this thesis, we show that it is indeed possible to statically enforce static (i.e. flat or hierarchical) RBAC policies and static categories in a CBAC policy. We propose, firstly, a static solution to RBAC policy enforcement for Java programs through the use of new *RBAC MVC* design patterns, which integrate roles into the program as a set of MVC components (i.e. classes) for each role. This methodology is then adapted for a hybrid solution to CBAC policy enforcement. Our new *CBAC MVC* patterns integrate

static and dynamic categories into the program as a set of MVC classes but also include patterns for a run-time reference monitor and the session. Each role's/category's associated MVC classes act as a role-/category-specific interface to accessing resources - the invocation of protected methods in resources are made in these role/category classes only.

In the first system, the flow of the program directs users to the set of role classes associated to their active role, therefore they should only be able to reach those protected method invocations that are made in the role classes associated to their assigned roles. Crucially, this enables the identification, at compile-time, of the role which is the owner of a method invocation. Finally, the protected invocations are checked statically for policy compliance. We present a static verifier, which performs syntactic checks and method invocation analysis to ensure the invocations to methods in resource classes are made only in role classes, and role classes do not invoke methods of components belonging to other roles. To the best of our knowledge, this is the first static verifier available for RBAC policies.

In the second system, the flow of the program directs users to either of two possible paths. The first is to the interface of one of their assigned static categories, from which they can access interfaces of dynamic categories which are related to the chosen static category and then from there to the interfaces of dynamic categories related to the current dynamic category, and so on. The second path is to other areas of the program which do not access resources. We present an enforcement mechanism, adapted from the first system, described at a high-level. It ensures each protected invocation (i.e. access request) made in each static/dynamic category's MVC classes abides by the policy. Finally, we show how code is generated into classes associated to dynamic categories to check at runtime if access requests should be granted or denied depending

on the user-to-category assignments in the policy (i.e. if the user is computed to be a member of a category permitted to invoke the action).

To the best of our knowledge, this is the first hybrid verifier available for general policies specified using CBAC.

1.6 Limitations and Restrictions

Our static program analysis is applicable to RBAC and CBAC implementations under certain important conditions. The first of these is that the source code must be available at compile-time. Secondly, the code should not be modified at run-time through mechanisms such as reflection, therefore our system is aimed at non-malicious, 'honest', programmers. Additionally in our first system, the RBAC policy should not rely on dynamic information (which changes throughout execution). The latter condition holds for the first and second 'levels' of the standardised RBAC models: *flat-RBAC* and hierarchical-RBAC [22] . This restriction is relaxed in the second system utilising CBAC. Note that due to using the static approach, any administrative changes to the policy after compilation, such as adding/removing/modifying roles/categories, would require the code of the program to be modified and thus recompiled. This would, however, ensure that errors are caught in the updated program code.

Lastly, note that the approach as presented in this thesis does not have formal semantics, properties or proofs. This is a result of time limitations, but is a very important and interesting aspect of this work and has thus been left for future work.

1.7 Overview of the Thesis

Part I reviews recent related literature and describes the necessary background concepts as follows. The first chapter contains the literature review and the second discusses access control, which includes the major models and enforcement mechanisms. The third chapter covers software engineering for security which includes Object-Oriented programming, design patterns, policy syntax and Java. Part II describes the first of our systems, which targets the static enforcement of RBAC policies. The first chapter presents the concept of the approach, as a high-level overview, which is followed by a chapter outlining our policy language for static RBAC policies. The next chapter specifies our new design patterns - RBAC MVC - for target programs to follow in the static case. This is followed by a chapter describing the static enforcement mechanism, which includes the policy and program parsing phase and the static checking phase. The end of this chapter, along with the following final one, describe the implementation of the verifier and a case study. Part III describes our second system, which targets the hybrid enforcement of CBAC. This part follows the closely the structure of the previous part for the static case. However, the fourth chapter describing the hybrid mechanism includes details of the code generation phase for the dynamic checking of dynamic parts of the policy. Along with this, the fifth and final chapter describing the case study also includes an example of the code generation phase. Part IV describes the implementation of our approach and evaluates it. Part V states the conclusions from this work and outlines the future work.

1.8 Publications

A paper on the first system, the static enforcement of RBAC, has been published [3] in the proceedings of the 10th International Workshop on Automated Specification

and Verification of Web Systems 2014 (part of the Vienna Summer of Logic 2014). This paper describes all aspects of the static system. A short paper on the second system, hybrid enforcement of CBAC, has been published [2] in the proceedings of the 10th International Workshop on Security and Trust Management 2014. This paper describes, in short, the concept of the approach and extensions to the previous static system. Also, a journal paper giving full details of the hybrid system is currently in preparation.

Part I

Literature Review and Background

Chapter 2

Literature Review

We begin by stating that formal approaches for the verification of properties of access control policies usually rely on purpose-built logics or rewrite-based techniques [12, 46, 42, 9]. In this thesis, we have focused on verifying that a program enforces a policy, rather than on proving properties of the policies. Focusing on the enforcement of policies, below we discuss the differences between our approach and several key related approaches.

2.1 Runtime Monitors and Hybrid approaches

The majority of reference monitors (see, for example [20]) enforce policies using only dynamic checks. This is the case for [18], where de Oliveira et al. present a systematic method to enforce a rewrite-based access control policy by weaving the policy into the program resulting in a program implementing a reference monitor. Their approach does not utilise static checks since they do not separate the static parts of the policy to enforce them statically.

A notable recent example of the hybrid approach is by Bodden et al. [11], where they enforce security properties in programs using a hybrid approach. The kinds of security properties they target are more general than only just access control policies. They generate code for run-time checks, then perform a series of compile-time analyses to remove some of these by performing those ones at compile-time. However, the access control enforcement of static categories would not be possible at compile-time. This is because they cannot determine, at compile-time, the access requests that each static category can make. Our design pattern solves this. Therefore, in their approach, a CBAC policy would be enforced using only run-time checks.

2.2 Web Security and Java

There is a line of work on utilising static analysis to identify vulnerabilities in web applications, then inserting code at the vulnerable points which perform run-time checks to prevent these from occurring. In [35], Murata et al. present a framework for the hybrid enforcement of access control policies in XML document query applications. The approach uses automata for representing and comparing the queries, access control policies and XML schemas. Access requests which are found to be denied are rewritten as empty lists at compile time, and those which cannot be evaluated to grant or deny at compile-time are checked at run-time. The idea of this work is very similar to ours - to evaluate every access request at compile-time if possible, or at run-time if not. They show how a hybrid enforcement approach can provide significant optimisations and assist programmers in creating their programs taking into account the access control requirements. Their work, however, targets specifically XML-based applications whereas we target OO languages.

Sun et al. propose in [49] an approach for detecting access control vulnerabilities in web applications. Their work focuses on finding cases where users can access pages which they do not have the right to access according to their role. They automatically detect a set of known vulnerabilities, which circumvent static RBAC policies, via static analysis, however we enforce general RBAC and CBAC policies via a hybrid approach.

In [30], Huang et al. use a hybrid approach to secure web application code. They protect against specific information flow vulnerabilities in web applications (using language such as HTML, Javascript, etc.) by identifying them at compile-time and protecting against them at run-time using inserted code. This kind of automatic analysis is more user-friendly than careful configuration of application-level firewalls; they limit validation to vulnerable sections of code rather than the whole application. We utilise the same idea of using static analysis to identify points where code needs to be inserted for run-time checking. However, our enforcement is also applied at compile-time, not just at run-time.

There are two types of in-built Java web security mechanisms, declarative and programmatic, which are both dynamic enforcement mechanisms, whereas our approach is hybrid. Most approaches based on Java utilise these mechanisms. Declarative web security is used in web clients, where Servlets and JSPs can be grouped into domains, then the programmer specifies the access rights (i.e. roles) needed to access each domain. When the user accesses a Servlet or JSP which is in a domain, they are dynamically shown a log-in page, where after successfully logging-in, they must have the access rights required by the domain to continue to interact with the Servlet or JSP. Otherwise, they are directed to an error web page. This log-in prompt is a dynamic mechanism - it is shown at run-time whenever the user requests access to a protected Servlet or JSP. Programmatic security can be used in either a web client or an application client. It is used to create dynamic conditional statements (e.g. *if* statements)

that check if the user, at run-time, has the necessary access rights to execute certain lines of code. JEE provides a security context implementation which holds the session-scoped dynamic information, which is consulted during these dynamic checks. Our hybrid approach targets application clients, in which we statically verify the parts of a policy that are static, and the dynamic checks do not fully rely on Java EE's in-built programmatic security API. Instead, it relies on the categoriser (i.e. reference monitor) that the programmer provides.

2.3 Model Driven Systems Security Engineering

There has been significant developments in the area of security at the design level. The general approach in this area is to specify security restrictions at the design stage of a program's lifecycle. One such approach is described by Basin et al in [6] and [34], where they propose Model Driven Security as an approach to building secure systems. In their approach, security policies are specified at the design level using UML notation, via a new language called 'SecureUML', on a model of the system, which is then transformed into an application with security enforcement code generated for a specific implementation platform. They give an example of a static RBAC policy which is transformed together with the target program into an EJB JEE program. The security code generated utilises Java Web security mechanisms. These mechanisms are dynamic (as discussed above), therefore the key difference between approaches like these and our approach is that we enforce policies using a combination of static and dynamic checks. The work by Spanoudakis et al. [45] is an example of static verification using UML. Their work focuses on the enforcement of more general of security properties (such as integrity) via model checking. Our approach differs in that our design patterns produce a structure to guide the flow of the program which enables static verification via code

analysis. Also, our approach targets access control security (rather than properties of security) which is enforced not only statically, but in combination with dynamic checks.

2.4 Type Systems for Security

There is a body of work enforcing security via type systems. Pottier et al. [39] propose an approach to statically perform Java's dynamic in-built access control verification known as *stack inspection*. Although their approach is specifically targeted towards stack inspection, conceptually the approach could, perhaps, be generalised to deal with wider access control policies to be enforced in Java applications. Our verifier can be formally presented as a type system and developing this could be inspired from their approach. The key difference in our approach is that access control is enforced dynamically as well as statically, rather than only statically.

In [36], Nanevski et al. statically enforce, via dependent types, several kinds of policies (including declassification, information erasure, state dependent access control and state-dependent information flow) combined together. They show how these policies can be expressed using types and enforced in programs. The static enforcement takes into account more dynamic features of programs such as dynamic allocation and deallocation. Their work targets information flow but also enforces state-dependent access control, whereas we target RBAC and CBAC policies and it is not clear how such policies can be specified and enforced in their approach. Also, they assume that the user which is making an access request can be retrieved from a local context, whereas we attempt to compute the user(role) at compile-time via combining our static analysis with design patterns which implement the policy features into the program.

An interesting line of work that can mitigate the problems of exclusive static type checking, as is done in [39], is *soft typing*. This is shown, for example, by Cartwright

and Fagan [15]. Soft typing performs as much of the type-checking as possible statically, with the remaining checks, that cannot be done statically, done at run-time. We intend to present our approach formally and thus it could be presented as a type system. Then, soft typing would be one of the most relevant approaches to adapt to our hybrid enforcement of access control approach. We have, as yet, not seen an example of soft typing being applied to the enforcement of access control policies in OO programs.

2.5 Security Patterns

There exists a body of work on security patterns. In [40], the authors describe access control, specifically RBAC and Metadata-based Access Control, using patterns and run time checks. However, the overall approach of our patterns (in the second system) is towards implementing CBAC via hybrid checks in JEE applications. Steel, Nagappan and Lai [47] propose several security patterns that are specifically targeted towards securing JEE applications. Our overall goal is to enforce CBAC in JEE applications using a combination of static and dynamic check. This significantly reduces performance costs associated with purely dynamic checks and provides the advantages of static verification such as easier debugging and static guarantees of the runtime behaviour of the program. However, their work takes a dynamic approach to enforcement. Many of their patterns can and should be used in conjunction with our patterns to secure the overall application aside from enforcing CBAC using a hybrid approach, such as the Intercepting Validator to properly check parameters before executing transactions and Secure Pipe to properly secure the connection between Client and Server or Server and Server. We have not seen any examples of patterns being used to aid the enforcement of access control policies via static code analysis (and code generation).

2.6 Annotation-Based Approach

Zarnett et al. [51] enforce RBAC in Java using an annotation-based approach on proxy objects in RMI. Their work has the effect of removing the need for run-time access control checks, however their approach relies on annotations, which has several weaknesses. Understanding where a specific annotation should go can be a difficult task, especially in large programs. Moreover, specifying the policy via annotations leaves the policy fragmented throughout the program. In our approach, we can check that the policy has been implemented correctly, e.g. that all the resources and roles have been implemented, however they have no such verification techniques since there exists no central policy specification, which means that errors are discovered later (at run time). Also, recalling the left-hand side of the model in Figure 5.1, their approach enforces access restrictions at the level of the 'Resources', by creating proxy objects containing only those methods which are authorised for the currently active role. Our approach enforces access control at the level of 'Tasks', where instead of creating proxy objects of each resource for each role all authorised methods for each role are provided by a user-interface in a role object. Another key difference is that we assume a system where server and client code are available and can be checked at compile time. A significant advantage of our approach is that it enforces a CBAC policy, whereas their approach is restricted to RBAC. Since their system relies on the in-built RBAC mechanisms that Java Security provides, it may be difficult to adapt their approach to CBAC. Our approach can be applied not just to RBAC, but to any of the models that CBAC can be specialised to, of which there are a many (some of which are demonstrated in [5].

2.7 Policy Specification and Verification

Whilst the novel contribution of our work is not the specification or verification of policies, it is nonetheless important to describe the advances in these areas and how they relate to our work.

In the literature, there are many examples of static verification of policies in order to determine their validity. The key difference in our work is that we present an approach for the enforcement of policies in programs, rather than the validation of properties of a policy such as correctness and completeness. Some of these examples are discussed as follows. Shafiq et al propose in [44] an approach for verifying the consistency and correctness of event-driven RBAC policies, using a coloured Petri-Net [32]. Their work deals with dynamic features of role based systems and our work also deals with dynamic features of policies but targeting more general dynamic features which can be expressed using CBAC. In [31], Hwang et al. present a tool for modeling and verifying access control policies called ACPT. As with our work, they utilise a combination of static and dynamic analysis. However, both of these are for the verification of the policy, where the static analysis verifies the policy against a set of user-specified properties which is complemented by dynamic verification through the generation and execution of test suites. In [1] Ahmed and Tripathi present an approach for the static verification of security requirements in role-based Computer Supported Cooperative Work (CSCW) systems. Their static verification ensures completeness and consistency of security requirements. This involves a formal specification model for security policies and development of a verification model to ensure that the requirements are correctly realised by a given design specification. The verification is applied at the design stage via model checking, whereas we analyse the code of an application which must follow our proposed design patterns. In [38], Pistoia et al. present an approach for the validation of enterprise security policies. Their approach includes a formal validation

model of RBAC and a static-verification method for checking properties such as the role requirements of an application, inconsistencies in the roles in the policy and redundant roles in the policy. The key benefit of this work is to help identify flaws in policies by pointing to points in the code where such flaws materialise, then to suggest improvements that can be made to mitigate the flaws.

One major example of a shift away from traditional access control models and policy languages is [23], where Fong presents an access control model and policy language for Relationship-Based Access Control (ReBAC). This model is a departure from traditional approaches, such as RBAC, since it focuses on the relationships between the members in a social network and expresses access control in terms of these relationships. The work serves as initial evidence of the applicability of ReBAC as a general-purpose access control paradigm for general information systems. The approach consists of a new protection model specified formally as a state transition system, as well as a policy specification language utilising modal logic. Our approach utilises RBAC and is focused on the hybrid enforcement of access control aided by design patterns. However, the work does not deal with how the policy would be enforced in a program, which is the focus of our work. A hybrid technique could possibly be used to enforce ReBAC policies, which would require identifying static and dynamic parts of such policies. For example, there may be permanent, fixed relationships, (perhaps familial relationships such as "mother", "father", "daughter", etc.) which could be static aspects able to be enforced at compile-time.

Chapter 3

Access Control

Access Control is the task of making sure a user or process accesses only those resources which they are authorised to [17]. We can see that this will require two things: to be able to distinguish the user or process requesting access, and to check if they have the right to access the resource.

Briefly defined below are some key terms and concepts which are commonly used when discussing access control.

- **Subject** or **principal** or **user**: An active entity, generally in the form of a person, process, or device that requests access to resources (see below) or changes the system state [7].
- **Object** or **resource**: An entity that contains or receives information. Access to an object usually means access to the information which that object contains. Typical objects are records, fields (in a database record), blocks, pages, files, directories, directory trees, processes and programs, as well as processors, video displays, keyboards, clocks, printers, and network nodes. Devices such as

switches, disc drives, relays, and mechanical components connected to a computer system may also be included in the category of objects [7].

- **Action**: An operation that can be performed on a resource e.g. 'print' on a resource 'printer'.

- **Task**: An active process invoked by a subject on a resource; for example, when an online banking user successfully logs in, the control program operating on the user's behalf is a process, but the subject can initiate more than one task such as a balance enquiry or a bank transfer [21].

- **Permission** or **privilege** or **authorisation**: The right to perform some action in the system on resources. The term usually refers to some combination of resource and action, where that particular action is allowed on that particular resource. A particular action used on two different objects represents two distinct permissions and, by the same notion, two different operations applied to a single object represent two distinct permissions. For example, a bank teller may have permissions to execute debit and credit operations on customer records through transactions, while an accountant may execute debit and credit operations on the general ledger, which consolidates the bank's accounting data [21].

- **Result**: The decision that is reached after evaluating an access request. The possible results of access requests are usually grouped into a set, which include common results such as *grant*, *deny*, *grant if* a condition is met and *undefined* (usually used in a distributed access control system).

- **Access Control Policy**: The high level rules specifying how access is managed [41]. It describes the user and the circumstances in which access may be granted to the information or resource.

- **Access Control Mechanism**: Done at hardware or software level, this is where the access control limitations described in the policies are implemented [41].

3.1 Access Control Models

An Access Control Model includes a description of the main entities in the system (usually subjects, resources, privileges) and the way access to resources is controlled. Roughly speaking, an access control security policy describes a specific system, with a set of users and their access privileges. Formal specifications of access control policies allow proof of properties on the security provided by the access control system being designed [41]. Thus, they are useful for proving theoretical limitations of a system. Access control models bridge the gap in abstraction between access control policy and access control mechanism [29]. Access control mechanisms are designed to remain faithful to the properties of the chosen model.

There are several models for access control. Below, the two historically major models are briefly discussed, followed by the widely-used Role-based and the meta-model Category-based Access Control.

3.1.1 Discretionary Access Control

Discretionary Access Control (DAC) is where the owner of each resource (files/data) decides what the access rights to the resource are. In this model, there is the concept of every resource having an owner, and the owner is given a certain amount of access control. This means that it is the owner who specifies the permissions (such as read or write) to the resource. Often, discretionary access control models are implemented using access control lists (ACLs), or an access control matrix.

- **Access Control Matrix**: A table in which each row represents a subject, each column represents an object, and each entry is the set of access rights for that subject to that object. The access control matrix can be represented as a list of triples, having the form <subject, rights, object>. Searching a large number of these triples is inefficient enough that this implementation is seldom used [48]. The matrix is usually subdivided into columns (ACLs) or rows (capabilities).

- **Access Control List**: A list linked to an object which specifies all the subjects that can access the object, along with their rights on objects [29]. Each entry in the list is a pair consisting of subject and set of rights. One column in the Access Control Matrix represents one ACL. They are most usually implemented directly into systems, or approximated as in most modern operating systems.

3.1.2 Mandatory Access Control

Mandatory Access Control (MAC) is where access policies are determined by the system with multiple levels of access. In this model, access control is regulated and enforced by a central authority, and not by owners of objects and so they cannot change the access rights. There are two prerequisites for when to use this model: when the access control should not be decided by the owner of an object, and when the system itself has to enforce security policies, not the object owner.

This model is usually multi-level, where access rights are divided into levels, subjects and objects are given a level, and subjects can only access objects on the relevant level. Subjects are active entities or processes that request access to objects, and objects are information stores [41]. An example use of multi-level MAC is a generic military system, in which there will be levels such as secret and top-secret, and subjects will only be able to access objects at some of those levels. Levels are implemented as labels.

This relates to the mechanisms which are usually used to implement MAC policies. These include the Bell-LaPadula rules:

- no read up, which means users running a process of a certain level cannot read a file at a higher level e.g. a user running a process with secret clearance cannot read files with a label of top-secret [29].
- and no write-down, which means users running a process at a certain level cannot write to files below that level e.g. a user running a process at top-secret clearance cannot write or create files with a label of secret [29].

3.1.3 Role-Based Access Control

Role-Based Access Control (RBAC) is a popular mechanism to protect resources from unauthorised use in an organisation, where each member has a role, or multiple roles, assigned to them [21]. A role is associated with a set of permissions for performing actions on resources and is directly related to a user's job function. It has been defined as a set of actions and responsibilities associated with a particular job in the organisation. Instead of specifying all the accesses each user is allowed to execute, as had been the case in the two previously discussed models, access authorisations on objects are specified for roles. Each role is given access rights, and each user is given roles so that only authenticated users who have activated the required role can access and use the restricted resources.

User membership into roles can be taken away easily and changing a member's role can be established as job assignments dictate. Role associations can be established when new operations are put in place, and old operations can be removed as organisational functions change over time. This makes it much simpler and easier to manage access control because roles can be updated without updating each user's access individually.

An example of RBAC use is within a hospital system, where one role could be that of a doctor, who may have authority to perform a diagnosis, prescribe medication, and order laboratory tests [29]; another role may be a nurse, whose authority can be limited to creating and editing patient records, and scheduling scans.

The concepts thus far correspondent to the lowest of the proposed standardised RBAC levels, known as *core-RBAC* or *flat-RBAC* [22]. In the next highest level, known as *hierarchical-RBAC*, roles can be arranged in a hierarchy, where a more senior role 'subsumes' another; the senior role inherits the permissions of the subsumed role and can be assigned further permissions. Role hierarchies are partial orders [22]; a transitive, reflexive and anti-symmetric relation. They correspond naturally to the way an organisation is structured thus improving the structure of, and reducing the effort required for, a RBAC policy specification.

RBAC has been the subject of extensive research, which led to widespread support and adoption in many systems including Microsoft SQL Server, Solaris and Java. [24] shows how Role-Based Access Control can be implemented in Java JDK 1.1, and the current Java Platform 2 Security [25] features APIs for implementing policies, which can be used to implement RBAC.

3.1.4 Category-Based Access Control

RBAC use had become widespread, but as technology rapidly advanced, especially the internet, many organisations and researchers found limitations with the model. There had been a shift in the way IT systems were designed. Specifically, there was a shift away from static systems, where users are known prior to creation of the system and data is centralised to one local location, to more dynamic systems, where users may not be predefined or known and data is decentralised. Traditionally, systems were also

designed with one central authority which designated the access privileges. However, modern network-connected, highly dynamic and distributed systems do not have this. There was no standard model which supported the dynamic, decentralised needs of modern systems, and so this became the subject of much research [4] [8] [16]. Further to these limitations, a permission in RBAC could not utilise any dynamic information e.g. time. These problems led to the creation of many new models, including extensions of RBAC. As a result of the large number of models being created, Barker created a meta-model of access control [5], which we call Category-based Access Control (CBAC). This is briefly discussed as follows; refer to [5] for a more detailed description.

CBAC has a small set of abstract, countable sets and the relationships between them can be specialised so that other models for access control are seen as instances of CBAC. There are familiar primitive notions of a countable set for each of *resource identifiers*, *actions* and *principals* (which we will interchangeably refer to as *users*). A permission is a pair (a, r) where action a can be performed on resource r. Next, there is the notion of a set of *categories*; a more general kind of group or classification to which a principal can belong to. Permissions are assigned to categories. Next, there is a set of *situational identifiers*, for environmental information, and a set of *event identifiers*. We refer to the latter two as *dynamic information*. Principals are assigned categories depending on rules specified by the policy author using (if necessary) situational or event identifiers. Every set is abstract and needs to be specialised for the application domain. One or more principals can be assigned to a category without dependence on situational or event identifiers i.e. principal p is assigned category c. This means that the principal-to-category assignment is static, and if all such assignments to a specific category are static, then the category itself is static. We call this a **static category**. This is equivalent to a role in hierarchical-RBAC, where principals/users are assigned to roles and no dynamic information is part of the policy definition. In addition, there is

a finite set *Auth*, of possible answers to an access request. CBAC is a highly expressive meta-model which can be specialised to a large number of other models [10].

3.2 Enforcement: Reference Monitor

The traditional means for enforcing access control policies is via a *reference monitor*. This is a program which monitors the execution of a target program, intercepting each access request at run-time, and checking if that access request should be granted or denied according to the policy. If the access request is denied, the target program is halted and rolled back to a safe state. Many access control models and policy languages rely on this traditional mechanism, to this day. It is, however, not the most efficient or effective mechanism, due to the fact that it costs run-time memory, because it is a separate program, and processor time, since it intercepts and evaluates every access request at run-time. It is also difficult to analyse and guarantee that the enforcement enforces the policy properly. Lastly, implementing and debugging this mechanism can be very difficult because the entire mechanism takes place at run-time. Debugging run-time errors requires rigorous run-time testing and then tracing the error requires going back to the code to find the error using the run-time bug information.

3.3 Language-Based Security

Programming techniques - such as language design, type systems, automatic program analysis, compilation and program rewriting - have been applied beyond their original scope to the area of security [43]. Originally, the operating system was solely responsible for enforcing the access control policy in the system using the reference monitor approach. Only the operating system kernel constituted the *minimal trusted computing*

base (MTCB). The MTCB principle states that the smaller the piece of software we trust, the greatest probability that it has no errors. However, as operating systems and programming languages increased in number and complexity, enforcing the policy solely in the operating system became difficult. Moreover, in many cases the resources were not implemented by the operating system. As such, it cannot restrict access to them.

Language-based techniques allow us to overcome these issues, as well as the issues of the reference monitor, to enforce the policy effectively and efficiently. The vast amount of work in programming techniques enables us to create safe run-times for languages. A widespread example of a language-based technique is method access modifiers in OO languages such as Java, which use compile-time analysis to restrict the invocations of methods.

Chapter 4

Software Engineering for Security

4.1 Object-Oriented Languages

The key notions of class-based OO languages, needed in the access control domain are briefly discussed as follows. In OO languages, programs are broken down into a set of classes and objects are instances of classes. Classes contain a set of data fields (i.e. variables associated with the class), which objects usually populate with values, and a set of methods. Methods can be specified for invocation on objects of their declaring class, when the populated fields need to be used, or on the class itself when this is not the case.

4.1.1 Types.

A variable in a *typed* programming language has an associated type. A type is a grouping of values that are similar in their behaviour and/or semantics. Common examples of types include integer, string and Boolean. In some OO languages, a class itself is a type, therefore the type of an object is its class.

4.1.2 Modifiers.

Most OO languages provide keywords to be used in the method declaration header or variable declaration that provide a form of coarse-grained access control on method invocations or variable usage, known as *visibility modifiers* (or simply *modifiers*). These specify that a method/variable can be called/used in either any class, by using (usually) the modifier 'public' or only within the declaring class, by using the modifier 'private'. Languages may provide more or fewer modifiers.

4.1.3 Inheritance.

Many OO languages provide a relation between classes called *inheritance*. This is when one class can be derived from another, thus the former is called the latter's *subclass* and the latter is called the former's *superclass*. The subclass *inherits* the fields and methods of its superclass. Due to inheritance, the subclass can share the type of its superclass, as well as specifying its own type. This means that the type of an object of a subclass can be its superclass as well as its own class. This shared type behaviour extends to subclasses of the subclass or superclasses of the superclass. Additionally, the subclass can *override* methods from its superclass. This is when the subclass contains a method with the same signature (i.e. name, modifier(s) and parameters) but with a different implementation. In some languages, including Java, the overriding method may call on the superclass's implementation of the method.

4.1.4 Dynamic Dispatch.

The ability for classes to inherit from one another combined with subclasses overriding methods leads to the problem of selecting which implementation of the method to run at run-time. There are some cases when it is difficult to determine this *statically* i.e.

at compile-time, which class's implementation of a method to invoke. Therefore, it has to be selected *dynamically* i.e. at run-time. For example, consider a class A containing a method m, and its subclass B containing an over-ridden version of m. Now, in a separate class C, we can create an instance of B but store it in a variable var of type A. This is valid since B inherits from A and thus objects of B share both types. If, in C, there is an invocation $var.m$, the problem is to decide which implementation of m should be selected - the one in A or the one in B. The selection must be the one in the actual type of the object that is stored in var - not the type of this variable. In this trivial example, we know that B's implementation should be selected. However, if we add more subclasses of A which override m, then the number of possibilities increases and it is difficult to know at compile-time what is the actual type of the variable stored in var. Therefore, the selection is done dynamically. This is called *dynamic dispatch*.

4.2 Patterns

A *pattern* describes a particular recurring design problem that arises in specific design context, and presents a generic scheme for its solution [14]. Patterns are usually described using the semi-formal Unified Modelling Language (UML) notation, showing its constituent components, their responsibilities and relationships, and the way in which they collaborate. They are used by developers, who are not necessarily familiar with formal methods. The goal of patterns is to provide a mechanism to guide the implementation of a solution to a specific problem. Patterns originate from known good development practices, and can be directly used to aid development. A widely used template for describing patterns is the Pattern-Oriented Software Architecture (POSA) template [14]. From it, we use the following sections to present the patterns:

- *Name*: Each design pattern has a unique name.

- *Example*: A concrete example of the problem's existence highlighting the need for the pattern.
- *Context*: The context in which the pattern can be useful.
- *Problem*: A description of the problem that this pattern solves.
- *Solution*: A description of the classes and objects used in the pattern, how they interact with each other and a graphical representation in UML.
- *Implementation*: A description of guidelines to help in the implementation of the pattern.
- *Consequences*: The results, side effects and trade-offs caused when using the pattern.
- *Known Uses*: Examples where the pattern is used.
- *Related Patterns*: A description of which patterns solve similar problems or are somehow related to this pattern.

4.2.1 Model-View-Controller Pattern.

Our work utilises concepts from the Model-View-Controller (MVC) pattern. This pattern achieves separation of concern for user interaction [33], separating data processing from user interaction, allowing both to be modified independently. Data access, data-processing logic and data presentation are divided into three distinct categories: Model, Controller and View. The Model components contain the data or data-related logic, views present the data, and the controller processes events affecting the model or views. Figure 4.1 illustrates the relationships between the components of MVC.

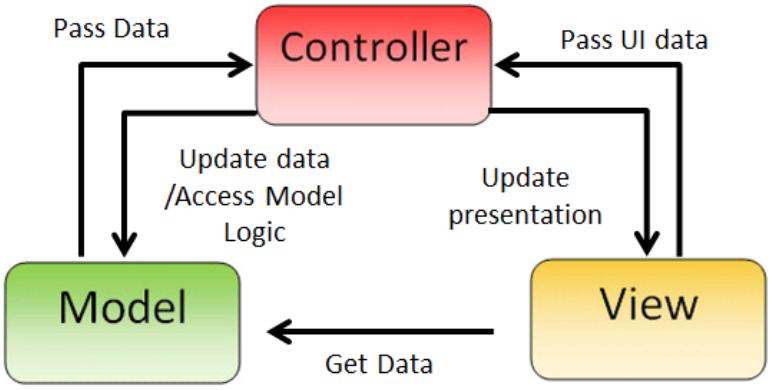

FIGURE 4.1: The relationships between components of the Model-View-Controller (MVC) pattern.

4.3 Policy Syntax

An access control policy is usually authored, in the first instance, by a *security administrator*. This is a person who knows the security requirements of the organisation to which the policy will apply. They are not required to have any computer science or mathematical knowledge - they are only concerned with the security needs of the organisation. As such, a policy written by a security administrator may not necessarily be done so by using the syntax specified for the wide range of access control models and languages in the literature. It may be written completely in natural language. There is a significant gap between the security administrator's version of the policy and the version to be applied in different contexts during an application life-cycle. For example, when developing an application for an organisation, the policy will need to be enforced in that application. Therefore, it will need to be adapted from the security administrator's version, which applies to real-world resources and users, to a version that can be applied directly to the program, which in the case of object-oriented languages consists of classes and methods. So the syntax of the policy to be applied in the program will be include resource definitions as classes and permissions as method

invocations on objects. This syntax can be very different to the syntax used in the security administrator's version, however it is important for the syntax to be specialised for the context in which it is needed. We choose to focus on a syntax specialised for implementation in the program. However, the same policy written in a language using a different syntax for a different context, could (and will need to) be translated to this syntax or a custom parser could be used to extract and transform the data generated from the policy.

4.4 Java Extended Edition (JEE)

Java EE is a platform for the creation of distributed applications using the Java language. It provides the means to create the server-side components and business logic of an application. These can then be used in two ways:

- an Application Client, which is a Java program on the client-side which interacts with server-side components through Remote Method Invocation (RMI) where methods are remotely invoked over a network, or
- a Web Client, where an internet browser on the client-side interacts with the server-side components through dynamic HTML pages.

In either case, JEE Applications usually follow an architecture consisting of three tiers, as shown in Fig 4.2 (taken from the JEE 7 tutorial). Although four tiers appear, the tiers are distributed over three locations: client machines, the Java EE Server machine, and database or legacy machines. For this reason, the JEE application model is considered to be a three-tiered architecture. We describe the main components of the architecture below.

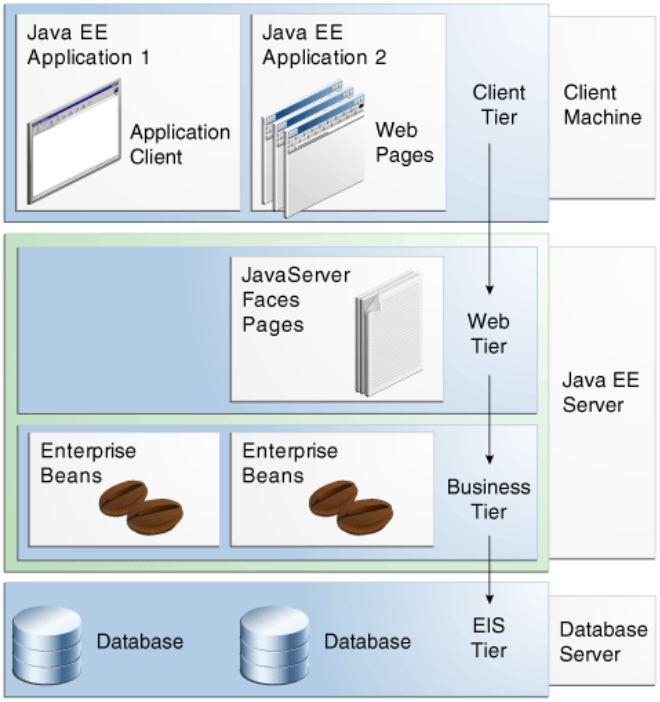

FIGURE 4.2: Three-Tiered Architecture of JEE Applications

4.4.1 Three-Tiered Architecture.

- The **Client Tier** consists of either a Java Application Client, or dynamic HTML pages which are generated by the server and accessed via a web client (usually an internet browser) on the client machine. These components run on the client machine.
- The **Server Tier** consists of two types of components:
  - **Web Tier** components of which there are three types:
    - Servlets, which usually control the flow of a web application and are invoked from web pages or JavaServer Pages (see below) via the web client,

    - JavaServer Pages (JSP) which are web pages interleaved with Java code, or
    - JavaServer Faces (JSF), which builds on Servlets and JSPs by providing a user-interface component framework for web applications.

    In this work, when discussing the Web Tier we focus on Servlets and JSPs, since they form the basis of a JEE application and JSF is actually built on top of these.

  - **Business Tier** components, called *Enterprise Java Beans* (EJB). EJBs are classes that contain the business logic of the application and are run on the JEE Server.

- The **Database Server Tier** consists of data sources such as databases, or other components such as web services.

4.4.2 Two Client Types, Two Cases

The two types of client, a Java Application Client and a Web Client, represent two cases for the structure of a JEE application since the Web tier is only used in the case of a Web Client. In this case, the client can invoke web tier components through dynamic web pages produced by server components. These may then call on EJBs to perform some business logic. In the Application Client case, the Web tier is not used. Instead, the EJBs are *injected* or *looked-up* in the client code, and then they are used as if they exist on the client side. The JEE runtime converts method calls on an EJB instance to RMI calls automatically. Application client usually use only Java code, whereas a web client can use a large number of web technologies and languages that have been developed e.g. JSPs JavaServer Faces, HTML and JavaScript and any of its libraries among others.

4.5 Java Security

JEE web security contains a means for RBAC to be implemented using dynamic access control enforcement. There are two types of access control using roles: declarative and programmatic security. Declarative security is a means for securing Servlets or EJBs using the application server hosting the application. Simply put, the programmer specifies the roles that are allowed to access specific Servlets/EJBs or methods within them, and when a user invokes these, the server performs authentication dynamically to check if the user has the required role. In programmatic security, the programmer specifies dynamic access checks within the code. They secure parts of the code by using provided methods such as `isUserInRole()` to specify that only certain roles can run that code.

Part II

Static Enforcement of RBAC

Chapter 5

Concept Overview for Static Enforcement

5.1 General Concept Overview

We propose a design methodology for OO programs, where RBAC is implemented at the 'Tasks' level (see Figure 5.1). Our goal is to statically enforce an RBAC policy that restricts the methods that users call on resources within a program. There are three main stages in this approach:

1. The first stage is specifying a policy following the RBAC model, where roles are assigned permissions to invoke methods on resource classes.

2. The second is designing programs such that they implement a specialisation of the general model of the flow of a program that implements RBAC (the general model is shown in the left-hand side of Figure 5.1). This is achieved through our design patterns, *RBAC MVC*, which we describe in more detail in the following

Chapter and in Chapter 7. Put simply, when users log-in, they are presented with an interface for the Session. This Session-specific interface then allows users to interact with the system via their role, performing their role-specific tasks in the system, or without their role thus performing "Other" tasks. Each role is implemented via a set of MVC components, which can be called by the Session interface when the user wishes to interact with the system using their currently active role.

3. The third stage is then running the static verifier, which checks the code in Role MVC classes to ensure that each method call to an access-restricted method respects the policy, and checks the whole program to ensure that calls to access-restricted methods are only made in these components.

5.1.1 Client Type: Application Client

Our focus for the type of client will be the Java application client. The reason for this is that in this case, only one language is used - Java. In the web client, multiple web languages can be used, particularly in the views. Since our verifier works at the source-level, if we chose to target the web client, we would need to develop a static verifier for each possible language used. This is a task that we plan to address in future work.

Below we give a more detailed overview of the second stage above.

5.2 Overall Program Flow and Static Approach

We now describe the general flow of RBAC programs and how this general flow is specialised in our approach to enable static verification.

In general, programs that restrict access to resources from users typically involve an initial user authentication phase, where users log in and retrieve their access rights, then allowing users to undertake user-tasks which may involve accessing resources, and finally logging out of the system. We present a simplified model of the general flow of a program which implements RBAC in the left-hand side of Figure 5.1. In RBAC, authentication also involves retrieving and activating the role(s) associated to the user, and logging out also involves deactivating the role(s). Controlling access most commonly takes place between 'Tasks' and 'Resources', for example through a reference monitor intercepting all access requests made to resources at run-time, stopping those requests which are invalid according to the policy. The left hand side of Figure 5.1 shows that each and every access request to resources must be intercepted (at run-time) by the reference monitor, to check if it is allowed. Only the allowed requests can then continue to access resources.

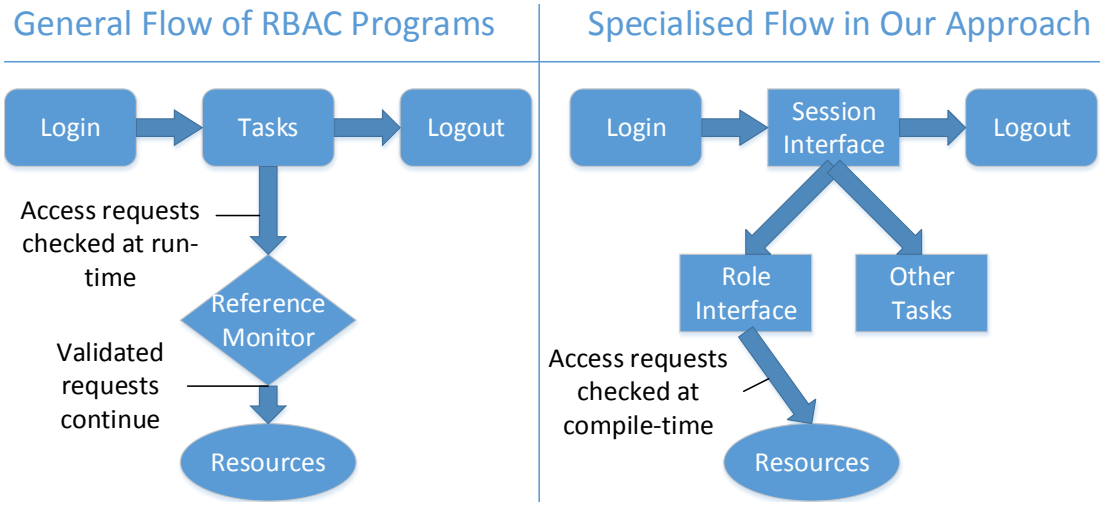

FIGURE 5.1: General and our Specialised flow of programs that enforce RBAC

In our approach, we divide user-tasks into three groups: 'Role tasks' that users with certain roles in the system can perform (these may access resources), 'Other tasks' that the policy is not directly concerned with and 'Session tasks' related to the functioning of the Session.

When at the 'Tasks' stage in our approach, there is a 'Session Interface' - which can be thought of as a "master" interface specific for the session. This contains components implementing Session Tasks, which will always be active, e.g. components for logging out which need to be accessible by the user at all times. Components for performing Role tasks and Other tasks are composed into the Session Interface. Role tasks are performed through 'Role Interface', which prevents direct access to the resources, as shown in the right-hand side of Figure 5.1. These are realised through a set of MVC components for each role: a set of View components, one Controller component and one Model component. A resource is realised as a Model component. In order to perform a task which may include accessing a resource, the user interacts with a Role View which will then communicate with its Role Controller, which will then communicate with its Role Model. Only the latter, Role Model, implements the business functions of the role - including tasks which may access resources. The Role Views deal with presentational logic for user-interaction and Role Controllers handle communication between Role Views and Role Models (and vice-versa). The implementation of 'Other tasks' is unrestricted in our approach, but will be verified to ensure the policy is not violated. The Session interface is also made up of MVC components, which handle three aspects: implementing Session Tasks e.g. log-in and log-out, linking to Role interfaces (to compose them with the Session Interface) and linking to the classes that implement 'Other Tasks'. In this way, roles are integrated into the program design, which ensures that it is possible to determine, at compile-time, which role is making a particular invocation. When verifying an invocation in a role component, the role which this component belongs to is the role that can reach this invocation. Our 'RBAC MVC' patterns guide the implementation of the program to achieve this flow.

Chapter 6

Policy Language for Static Enforcement

As we have already mentioned, the goal of our approach is to statically verify a Java program that contains some classes containing methods whose invocation needs to be restricted. This restriction is specified using hierarchical-RBAC. Thus, these classes become resources (in RBAC) and the restricted methods are the operations (in RBAC) that can be performed on them. In this chapter we first define the core concepts of the policy in the context of our approach, and then present a language that will be used to specify RBAC policies restricting access in Java programs.

Definition 6.1 (Resource)**.** A *resource* is realised as a Resource class containing some methods whose invocation needs to be restricted. Invocations are restricted for instances of Resource classes.

Methods in resource classes are categorised as follows.

Definition 6.2 (Actions and Auxiliary Methods)**.** An *action* is a method in a resource class that must only be invoked by those users with the permission to do so.

An *auxiliary* method is a method in a resource class that is not part of the policy definition. Such methods are usually required for the correct initialisation and operation of a class, and should not be invoked directly by users.

Definition 6.3 (Permission)**.** A *permission* is a pair $[res, act]$ where res is the name of a resource and act is the name of an action of that resource. The action is allowed to be invoked on any instance of that resource class by the role (see Definition 6.6) which the permission is assigned to.

Definition 6.4 (Access)**.** An *access to a resource* is an invocation or call to an action method of an instance of a Resource class.

Definition 6.5 (Task)**.** We divide the concept of a user-task into three groups as follows. Firstly, a *Role Task* is an operation, or business function, to be performed by an authorised user in a specific role, which could involve the invocation of one or more actions on resources. Secondly a *Session Task* is an operation required to correctly manage the session e.g. log-in and log-out. Thirdly, an *Other Task* is an operation or function that is executable by all users, regardless of the notion of role as it does not access resources (in the access control sense).

An example of a Role task is as follows. A user in an Admin role in a GP surgery may need to perform the task `registerPatient`, which would involve a call to an action e.g. `addPatient` in the `Patients` resource.

Definition 6.6 (Role)**.** At the policy level, a *role* consists of a name and a list of permissions to access resources.

In the context of our system, a role is implemented by a set of MVC components: a set of Role View components (i.e. classes), a Role Controller class and a Role Model class (as defined below). Together, these provide a *Role-specific Interface* for the user to perform tasks. We define these components below.

Definition 6.7 (Role Model)**.** A *Role Model* provides Role Task methods which should only call those actions that are permitted for its role. Its name must be prefixed with the name of the role, followed by 'Model'.

Definition 6.8 (Role Controller)**.** A *Role Controller* acts as an intermediary between the Role Model and View classes. Its name must be prefixed with the name of the role, followed by 'Controller'. Role Controller methods are invoked by Role View classes to communicate with the Controller.

Definition 6.9 (Role View)**.** A *Role View* provides (part of) the user-interface for users to execute the tasks of their role. Its name must be prefixed with the name of the role, followed by 'View' (followed by any valid Java identifier).

For any role r, its single associated Role Model class contains the code that performs the tasks r can do in the system. The role's multiple associated Role View classes and its single associated Role Controller class, provide the means for users that have activated this role to access these tasks (and perform them).

An example of a set of role components is as follows. Firstly, Role Model class *AdminModel*, which provides a Role Task `registerPatient()` that calls on the `addPatient()` action in the `Patients` resource. Secondly, a set of role view components *AdminViewPatients, AdminViewAppointments,* e.t.c. Thirdly, a role controller *AdminController* acting as an intermediary between the role view and model components.

Definition 6.10 (Session)**.** A *Session* is the state of the program in which an authenticated user is able to perform the three kinds of tasks in the system. The Session has a user interface composed of a Session-specific interface, the role-specific interface (made up of Role MVC components discussed above) of the current active role and any interfaces implementing Other tasks. The Session-specific interface is made up of a set of MVC components: one Session Model, one Session Controller and a set of

Session View classes. The Session Model implements the Session Tasks which are: log-in/authentication, role activation, log-out, calling a role-interface and calling classes that implement 'Other Tasks'. The Session Views and Controller provide the means for the user to access these Session tasks. Names of Session classes start with the string 'Session' followed by either 'Model', 'Controller' or 'View'. For the latter, since there can be many Session View classes, any valid Class identifier (in Java) is allowed to follow in the name.

The Session-specific interface should always be active so that the Session tasks are always available to the user. We, of course, have minimum expectations such as log-out is only available if logged-in and so forth. The Session-specific interface also allows the user to interact with the system via their role by calling a role-interface, or without their role thus calling Other task implementing classes.

The classes required for the session - Session Classes - constitute part of our Trusted Computing Base (TCB); the other part is the actions (within Resource classes), which we trust behave safely. The Session classes should contain the minimum code necessary to implement Session tasks, so that the TCB is small. We perform few checks, and exercise few constraints, on Session classes and actions, in order for their implementation to be as flexible as possible. Therefore, we do not deal with authentication in this thesis. However, an important aspect of an RBAC system is activating a role, which is to be implemented by the Session classes. We give guidelines for role activation in our approach below.

Definition 6.11 (Role Activation)**.** *Role activation* constitutes storing the chosen role in a field called *activeRole* in the `SessionModel` class and invoking the Role Controller of that role. This process is achieved in a method *'activateRole()'* in the SessionModel class (See Definition 6.10). This will result in presenting a Role View of the active role's Role-specific Interface to the user by composing it with the Session Interface.

6.1 Policy Language: JPol

We define a policy specification language for hierarchical-RBAC where resources, together with their associated lists of actions, and roles, together with their associated permissions, can be declared. To simplify, we assume that only the access requests that are allowed are expressed, so all other requests are not allowed. The policy file will be parsed and represented as a set of tables, to be used only at compile-time by the static algorithm in order to perform the access checks.

The policy language does not support user definitions and user-role assignments, since we do not deal with authentication in this thesis. Since with our static algorithm, only the roles which have been declared in the policy will be permitted to be assigned to users, the resources will still be protected because each role will have been checked at compile-time to ensure it does not perform any illegal access requests. The proposed approach is flexible: new users can be added to the system and role assignments can change depending on changes within the organisation.

6.2 Syntax and Representation

The policy language adopts an object-oriented, Java-like syntax designed to make the policy implementer's transition from target program language, Java, to policy language as effortless as possible. However, as we will see later, the static algorithm relies solely on the information generated as a result of parsing the policy file. Thus, the syntax of the policy language can change and be adapted to any environment using hierarchical- (or flat-) RBAC.

The grammar of the policy language is as follows, where the keyword *subsumes* indicates role inheritance. The abstract syntax of the policy language is illustrated in Figure 6.1.

```
stmts           ::=   (stmt ';')+
stmt            ::=   decRole | decRoleSubsume | decRes
                      | addActRes | addPermRole
decRole         ::=   'Role' ID '=' 'new' 'Role' name
decRoleSubsume  ::=   'Role' ID '=' 'new' 'Role' name
                      'subsumes' ID
decRes          ::=   'Resource' ID '=' 'new' 'Resource' name
addActRes       ::=   ID '.' 'addAction' name
addPermRole     ::=   ID '.' 'addPermission' permission
name            ::=   '(' ID ')'
permission      ::=   '(' ID ',' ID ')'
```

The Parser for the policy specification language checks that a policy declaration is syntactically correct, producing the Abstract Syntax Tree (AST) shown in Figure 6.1. It then generates intermediate data structures - tables called *'Resources'* and *'Roles'* containing the information needed for the static verification algorithm.

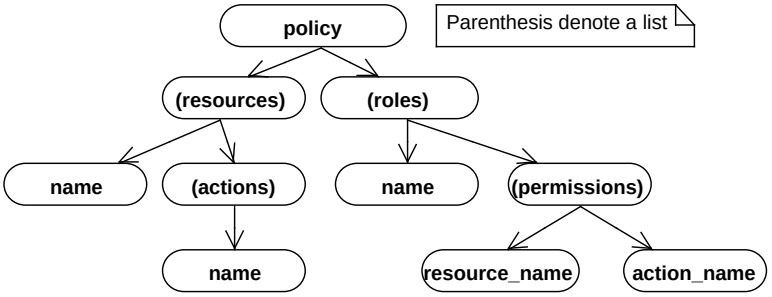

FIGURE 6.1: Abstract Syntax of Policy

Listing 6.1 shows an example specification in JPol for patient-related resources and permissions for roles in an example GP/doctor's surgery, with the resulting tables 'Resources' and 'Roles' shown in Figure 6.2.

```
Resource nhspatient = new Resource('Nhspatient');
nhspatient.addAction('getFirstName');
Resource privatepatient = new Resource('Privatepatient');
```

```
privatepatient.addAction('getFirstName');
Role nhsdoctor = new Role('NHSDoctor');
nhsdoctor.addPermission('Nhspatient', 'getFirstName');
Role privatedoctor = new Role('PrivateDoctor');
privatedoctor.addPermission('Privatepatient', 'getFirstName');
Role admin = new Role('Admin');
admin.addPermission('Nhspatient', 'getFirstName');
admin.addPermission('Privatepatient', 'getFirstName');
```

LISTING 6.1: Example JPol code declaring Resources with their actions and Roles with their permissions

Resources	
name	**actions**
Nhspatient	getFirstName
Privatepatient	getFirstName

Roles	
name	**permissions**
NHSDoctor	Nhspatient, getFirstName
PrivateDoctor	Privatepatient, getFirstName
Admin	Nhspatient, getFirstName
	Privatepatient, getFirstName

FIGURE 6.2: Example Roles and Resources Tables Representation

6.3 Semantics

We can state the semantics of the policy language in a concise manner by mapping the abstract syntax to elements of the RBAC model: there is a one-to-one correspondence between the resources, roles and permissions specified in JPol and in the RBAC model. In particular, an "addPermission" statement in JPol syntax (see the grammar rule for addPermRole above) corresponds directly to a permission in the RBAC sense. Therefore, we can define policy satisfaction as follows.

Definition 6.12 (Policy Satisfaction)**.** A Java program satisfies a JPol policy if, for any invocation $res.m$ that exists in the program, where res is an instance of a resource

class *Res* and m an action, only authenticated users with active role r, such that the JPol policy specifies the permission $[Res, m]$ for r, can perform $res.m$.

Chapter 7

Target Program Patterns

In order for the target program to be statically checked for policy compliance, it must follow our RBAC MVC Patterns described below. The goal of these patterns is to specify a scheme for implementing access to resources utilising RBAC concepts. For static checking, it must be known statically where in the code action invocations are allowed to be made, then which actions are invoked and then for each action invocation, the role that will make the invocation (at run-time). This is achieved by providing a model component for each resource and a model, a controller and a set of view components for each role. The role model component provides the tasks that users of that role can perform, and so invocations of actions are only allowed in these components. Each role's MVC components, together with the *Session MVC* pattern, provide a scheme for the system to operate such that users perform all their tasks, which may involve accessing resources, through their roles which are implemented as sets of role MVC components. The program can then be statically checked to ensure that action invocations only occur in role model components, and that the actions invoked are valid according to the policy for the role of each role model component.

7.1 Restriction due to Dynamic Dispatch

As described in Chapter 4.1, it is difficult to know statically for a method invocation the class in which that method is declared. This can cause problems in our approach. For example, if a resource class inherits from another resource class, then invoking a method on the subclass could, at run-time, actually be an invocation on the superclass. Another example is that a class which is not related to the policy or the session - an 'other' class - may inherit from a role MVC class. Therefore, an invocation on an 'other' class may, at run-time, be an invocation on the superclass. These possibilities can lead to policy violations which are undetectable at compile-time. To deal with this, inheritance is disabled between:

- two resources,
- two role MVC classes belonging to different roles and
- a role MVC class, a resource, a session class and and 'other' class.

7.2 RBAC Model Patterns

This pattern describes the design of Resource and Role model components. In this case, permissions on the resource are defined at the *class level*. This means that a role is assigned the permission to invoke an action on *any instance* of the resource. Its aim is to represent a resource and a role as classes such that the role classes present a coarse-grained set of methods which correspond to tasks, each of which may invoke one or more actions. Action invocations should only be made in these task methods.

- *Name*: RBAC Model

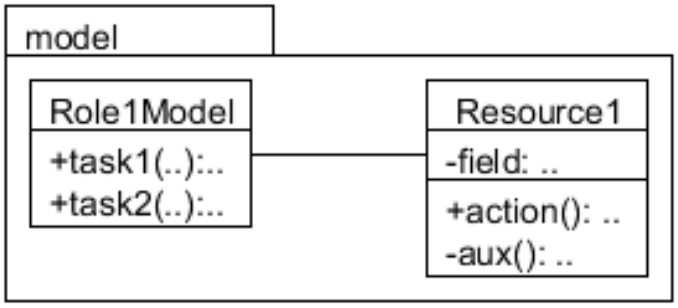

FIGURE 7.1: UML Class Diagram of RBAC Model Pattern

- *Example*: A system in which some classes represent resources, e.g. a class each for tables in a GP Surgery database such as *Patients, Appointments, Staff*. Accessing a resource is achieved by invoking an action method of a resource class, such as a method `findAll()` in each of the classes representing these tables. Access to each table needs to be restricted using the RBAC model, so different roles have a different set of permissions associated with them. Roles such as 'Admin' and 'Staff' would exist. Each role has a set of Role Tasks that it can do e.g. 'Admin' can do a Role Task 'Register Patient'.

- *Context*: An application in which access to resources needs to be restricted, following the hierarchical-RBAC model, with administrative changes disabled, and where access checking is to be performed at compile-time. In this case, the policy specifies restrictions on method invocations by roles. A resource is a class (e.g. PatientsTable) which contains some methods that can only be invoked by roles that have been assigned the permission to do so (e.g.`read()`, `write()`, `delete()`). We call such methods *action* methods. Any other methods in resources are called *auxiliary* methods. So roles are associated with a set of permissions; the right to invoke a particular action of a resource. Assigning a role the permission to invoke an action on this kind of resource means that the invocation can be on any of

its instances. At compile-time, the static verifier will decide if all invocations of actions in the program are valid according to the policy.

- *Problem*: The policy specifies permissions for roles to invoke actions on resources, however a user's interaction with the system will usually consist of performing a set of *tasks* in the system that are dependent on their role(s). The business logic part of the program needs to implement these tasks. Also, static verifier must ensure that the action methods can only be invoked by users with the required role and that the action invocations are valid according to the policy. In the business logic part of the program, when an action is invoked in the code, the static verifier must be able to know statically which roles can reach that invocation. Then it can be checked if those roles have the permission to call that action according to the policy.

- *Structure*: The structure for the solution is shown in the UML Class diagram in Fig 7.1. It contains:

  - *Resource1*: a Resource Model component, which is a class representing a resource. One class for each kind of resource in the policy (referred to as *Resource1, Resource2, .., ResourceN*) is needed. Named the same as the represented resource (without suffix 'Model').

  - *Role1Model*: a Role Model component, which is a class representing a role. One class for each role in the policy (referred to as *Role1, Role2, .., RoleN)* is needed. Named the same as the represented role with the suffix 'Model'.

  Note that there can be multiple packages containing Resource and/or Role Model classes but we do not restrict the names of the packages.

- *Dynamics*: The business logic part of the application contains Resource and Role Model classes (any classes not related to the policy can also be included).

Resource classes contain a set of action and auxiliary methods. Role Model classes contain a set of *task* methods which contain the business logic for carrying out tasks based on the role and can contain one or more action method invocations. In the business logic part of the application, invocations of actions should only be made in Role Model classes and Resource classes.

- *Implementation*: Resource Model classes must be named the same as the resource they represent. Using the same example, there would be classes `PatientsTable`, `ApointmentsTable` and `StaffTable`. Action methods should have the modifier 'public' and auxiliary methods should have the 'private' modifier, so that they are not accessed by other classes. If each of the classes had an action `findAll()`, it would have the modifier `public`. If each had an auxiliary method `format()`, it would have the `private` modifier. Role Model components should be implemented as classes with the same name as the role followed by the suffix 'Model' e.g. for a role 'Admin', the Role Model class name will be `AdminModel`. The Task methods should have the 'public' modifier. In JEE, the model components can be implemented on the server-side, as EJBs.

- *Consequences*: Roles and Resources are implemented in the program using Role Model and Resource classes, respectively. For the former, these act as the interface to performing tasks in the system, which includes accessing the Resource classes, so other parts of the program should not access resources directly. Where an action is invoked in the business logic part of the program, we know statically what role will be making the invocation - the role that is represented by the Role Model class in which the action is invoked. The static verifier can then check if that role has the permission to call that action according to the policy. Note that actions can be invoked in Resource classes also, which is left unrestricted (in terms

of both design and static analysis) in order to allow flexibility in implementing resources that depend on one another.

- *Known Uses*: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses static RBAC (i.e. permissions require no dynamic/run-time data). In this thesis, we show an implementation in JEE (see Chapter 9).

- *Related Patterns*: RBAC Controller, RBAC View, RBAC Session.

7.3 RBAC Controller Patterns

The following describes the design pattern for the controller component required for each role.

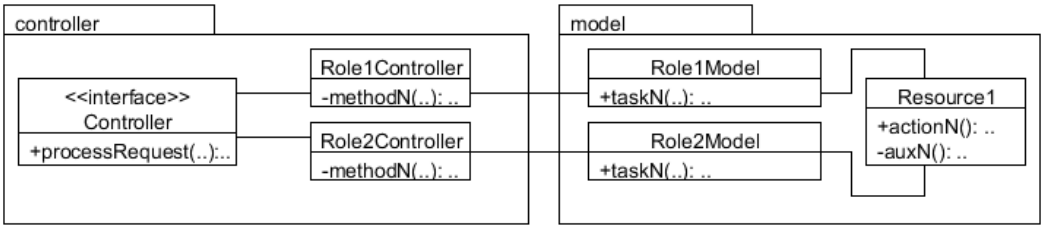

FIGURE 7.2: UML Class Diagram of RBAC Controller Pattern

- *Name*: RBAC Controller

- *Example*: The example is carried over from the previously discussed RBAC Model pattern.

- *Context*: The context is also carried over from the RBAC Model pattern, however the application is assumed to follow the RBAC Model Pattern described above.

So, each role has a corresponding model component, a Role Model class, which provides the Task methods which may invoke actions on resources.

- *Problem*: Each role has an associated model component, a Role Model class, which implements the business logic for the tasks which that role can perform in the system. Consequently, each role will also need a controller component to mediate communication between view components and the Role Model classes. The controller has two responsibilities. This first is to process input from the user from view components to decide which task(s) in which Role Model class(es) to invoke. The second is to process output from the Role Model classes to map the output to certain view components. Similar to the RBAC Model pattern, when an action is invoked in the controller parts of the code, the static verifier must be able to know statically which roles can reach that invocation. Then it can be checked if those roles have the permission to call that action according to the policy.

- *Structure*: The structure is shown in Figure 7.2. As well as the model classes from the RBAC Model pattern, it contains the following elements:

  - Package *controller.roles*: One or more packages containing role controllers .
  - *Role1Controller, Role2Controller*: These are *Role Controllers*. These are controller components corresponding to each role in the policy, named the same as the matching role suffixed by the string 'Controller'.
  - *RoleController*: A possibly empty interface, with this exact name, that each Role Controller must implement.

  Note that there can be multiple packages containing Role Controllers, as long as their names contain the string 'controller'.

- *Dynamics*: A Role Controller class provides the controller component that acts as intermediary between View components and the Role Model class of the same Role. It handles input from the views to select one or more Role Tasks to invoke, and maps the output of a Role Task to a view. Each Role Controller is associated with only one Role Model, the one with the same Role name. No other class can invoke a method in a Role Model class. Additionally, since Role Controllers indirectly invoke actions through invoking tasks, no classes except role-specific View components (discussed in the next pattern) should invoke Role Controllers. Role Controllers can also call actions, therefore any action invocation must be checked by the static verifier. The 'RoleController' interface groups Role Controllers so that they can be linked to as a group.

- *Implementation*: In JEE, there are two separate implementation cases. In the case of a Web Client, Role Controllers can be realised as Servlets, which contain the HTTP GET, and HTTP POST methods, implemented as `doGet()`, and `doPost()`. The latter two methods must be 'protected', as per the API specification. In the case of an Application Client, Role Controllers can be realised as normal Java classes acting as controller components.

- *Consequences*: Role Model components will be interacted with using role-specific controller components. In order to perform Tasks and access resources, the view components presented to the user must interact with these Role Controllers. Action invocations in controller classes can be checked statically since the role that would invoke that action will be the role that is represented by that Role Controller. Methods in Role Controllers have the 'public', 'protected' (as well as 'private') modifier, meaning they are accessible in other classes therefore their usage needs to be verified at compile-time by the static verifier.

- *Known Uses*: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and/or 'protected', where the access control policy uses static RBAC. in this thesis, we show an implementation in JEE.
- *Related Patterns*: RBAC Model, RBAC View, RBAC Session.

7.4 RBAC View Pattern

Now that we have defined role-specific model and controller components, we define role-specific view components.

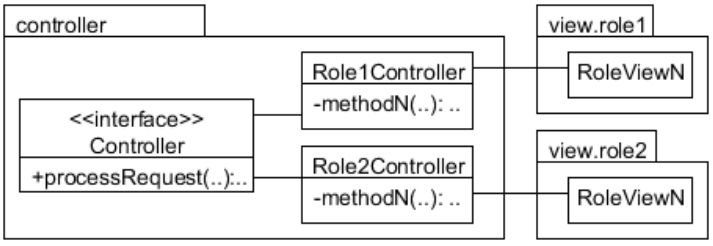

FIGURE 7.3: UML Class Diagram of RBAC View Pattern

- *Name*: RBAC View
- *Example*: The example is carried over from the RBAC Model pattern.
- *Context*: The context is also carried over from the RBAC Model pattern, however the application is assumed to follow the RBAC Model and RBAC Controller patterns discussed above. So, each role has a corresponding model component (i.e. a Role Model class), and controller component (i.e. a Role Controller class).
- *Problem*: Each role has an associated controller and model component. The UI presented to the user should allow them to perform only the tasks that are specific

to their role(s). The view components used to build and display the UI must interact with the Role Controller(s) associated with the same Role(s). Similar to the RBAC Model pattern, when an action is invoked in the presentation part of the code, the static verifier must be able to know statically which roles can reach that invocation. Then it can be checked if those roles have the permission to call that action according to the policy.

- *Structure*: The structure is shown in Figure 7.3. As well as the controller classes from the RBAC Controller pattern, it contains the following elements:
  - Package *view*: One or more packages for the view components for each role.
  - *Role1ViewN, Role2ViewN*: The view components that are specific to each role. Names are restricted such that they begin with the name of the role, followed by the string 'View', followed by any valid class identifier.
- *Dynamics*: Each role has an associated set of view components. These views present the UI elements needed for each specific role's tasks. Each Role View interacts only with the same role's Role Controller. The Role Views can invoke actions, for example to print a 'PatientsTable' object's data to the screen by calling its 'read()' method.
- *Implementation*: In JEE applications, there are two separate cases. In the case of a web client, the Role View elements could be implemented as JSP pages. Invocations to the same role's controller component would be achieved via a form or button submit action (or similar). This would invoke the Role Controller Servlet's HTTP GET or POST method, which could in turn invoke any other method in the Role Controller or a Role Task in the Role Model class belonging to the same Role. In an Application Client, they would be normal Java classes implemented as view components. To interact with the same role's controller component, Role Views will contain an invocation to the same role's Role Controller methods.

- *Consequences*: The presentation part of the application is separated from the rest of the program, and also segregated into groups reflecting the roles specified in the policy. Each role has a specific set of UIs. The user can only perform those operations in the system that are available in the Role Views for the role(s) currently active for their session. These can then be statically checked to ensure they only invoke the same role's controller component. Action invocations in Role Views can be statically checked since it can be known which role would invoke that action - the role that is associated with that Role View.

- *Known Uses*: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses static RBAC. In this thesis, we show an implementation in JEE.

- *Related Patterns*: RBAC Model, RBAC View, RBAC Session.

7.5 RBAC Session Patterns

The previous patterns have defined a scheme for the program so that the user interacts with the application through their role-specific MVC components. Now, we define a pattern that guides the implementation of the Session in our approach, which is a Session-specific interface implementing the Session Tasks made up of MVC components - one Session Model, one Session Controller and a set of Session View components. See Definition 6.10 for details of the Session Tasks and the behaviour of the Session interface.

- *Name*: RBAC Session

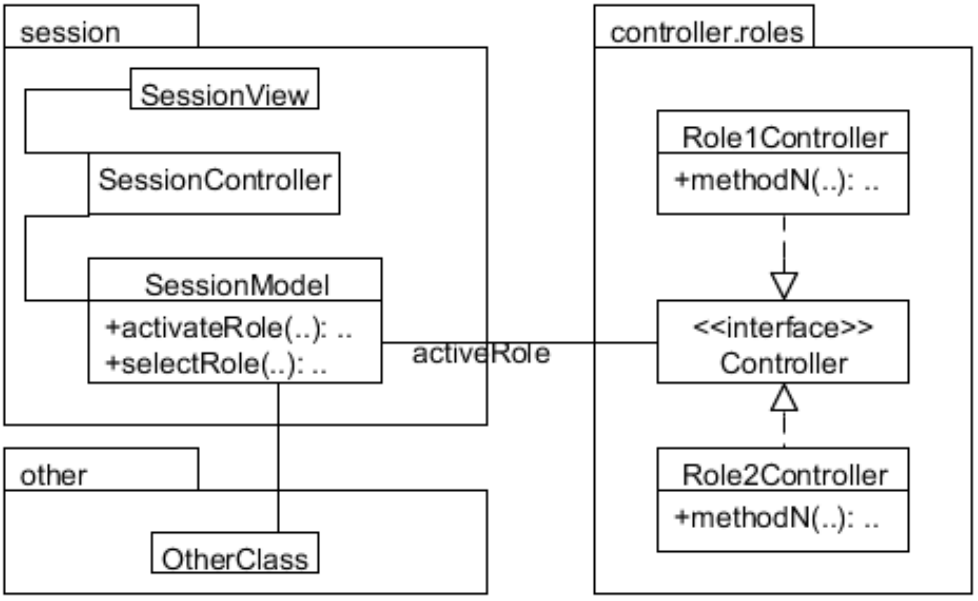

FIGURE 7.4: UML Class Diagram of RBAC Session Pattern

- *Example*: The example is carried over from the RBAC Model pattern. Specifically for this pattern, users have multiple roles assigned to them. For example, a user could have the role 'Admin' and 'Employee' assigned to them.

- *Context*: The program follows the RBAC Model, RBAC View and RBAC Controller patterns described above. Each Role has a specific set of MVC components which are used by the user in order to interact with the application.

- *Problem*: Each role has a specific set of MVC components associated with it. This means that each role becomes an interface to the application. The flow of the program should reach a stage in which the user can perform the three types of Tasks in the system: Role Tasks corresponding to their active Role, Session Tasks or Other Tasks.

- *Structure*: The structure for this pattern is shown in Figure 7.4. As well as the Role Controller classes from the RBAC Controller pattern, it contains:

  - *SessionModel*: A model component representing the notion of a session. It contains methods that implement the Session Tasks. Specifically, it links to the Role Controller (and also Role Views) of the currently active role of the user, to call that role's role-interface and display it to the user.
  - *SessionView*: One or more view components representing the Session, which displays a user-interface allowing the user to access and perform Session Tasks.
  - *SessionController*: A controller component representing the Session. The user-interaction from the SessionView is passed to the SessionController, which processes it and decides what to do with it e.g. invoke a method in SessionModel. It then retrieves the result from the SessionModel and displays it using a SessionView
  - *OtherClass*: One or more classes implementing Other Tasks.

- *Dynamics*: The user's interaction with the system begins through the Session components. These implement Session Tasks, which the user initiates through the Session Views, which communicate with the Session Controller, which processes the input from the Views and decides which Session Task to invoke from the Session Model class (if any). These Session MVC classes constitute a Session Interface. This interface is active at all times, to enable the user to invoke Session Tasks at any time. The user is initially presented with the Session View(s) that show the log-in/authentication user-interface components. If successful, the roles assigned to the user are retrieved and stored in the Session Model class. Then, the user can choose to interact with the system via their role, by activating one of the retrieved roles. This invokes the Role Controller of that Role, which then calls one or more of that role's Role Views which are composed with the Session

Interface. The user can also interact with the system without their role, thus invoking an Other class, which contains code implementing Other Tasks.

- *Implementation*: In JEE, the pattern can be implemented in two ways depending on the two cases. The Session Model would be similar in both cases i.e. a Session Bean implementing Session Tasks as its methods. When using an Application Client, the Session Views can be implemented as a Window e.g. *JFrame* and menu components e.g. *JMenuBar, e.t.c* which enable Session Tasks to be invoked. Role Views can be composed into the main display area of the window e.g. as a *JTabbedPane*. After authentication, the list of roles assigned to the user can be stored in a data member *retrievedRoles* in the Session Model. If the user selects the Session View to interact with the system via their role, the Role activation task in the Session Model would be invoked through the Session View passing the selected operation to the Session Controller, which would invoke the Role Controller of that Role. Logging out is left open to the programmer but we assume it clears Session View panel. When using a Web Client, the HTTPSession provides the notion of Session and thus no special class is necessary. The Session Views could be implemented as HTML menu items. The retrieved roles can be stored in the HTTPSession object as a key-value pair where the key is *retrievedRoles* and the value is a list e.g. *ArrayList* of role names. This is written to the HTTPSession object at the end of the authentication process. The Session View can be implemented as a fragment of a JSP/HTML page such as a sidebar menu which is included in all Role Views. The Session Controller can be implemented as a Servlet.

- *Consequences*: The user can switch between multiple retrieved roles where the view associated to each role presents a set of UIs unique to each role. Also, the

user can initiate Session Tasks via Session MVC components or invoke classes that implement Other tasks.

- *Known Uses*: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses static RBAC. In this thesis, we show an implementation in JEE.
- *Related Patterns*: RBAC Model, RBAC Controller, RBAC View.

7.6 Example Of Patterns

The patterns can be applied to many OO languages and implementation platforms, where there is the concept of visibility modifiers and packages. We show an example of implementing the patterns for the GP surgery running example.

7.6.1 RBAC Model

Figure 7.5 illustrates the RBAC Model pattern being used for designing the patient-related operations in the system. There are Resource classes for each of the 'NHSPatients' and 'PrivatePatients', where each instance will represent one row in the corresponding table. The table itself is representing via the '-Facade' classes, which deal with querying the table to retrieve or update data e.g. retrieving all rows in the table as a list of instances of the corresponding class that represents a row. There are also three Role Model classes corresponding to the roles 'NHS Doctor', 'Private Doctor' and 'Admin'. The structure reflects the policy, which can be easily gleaned from the diagram - note that both the row classes and table classes/facades are Resources.

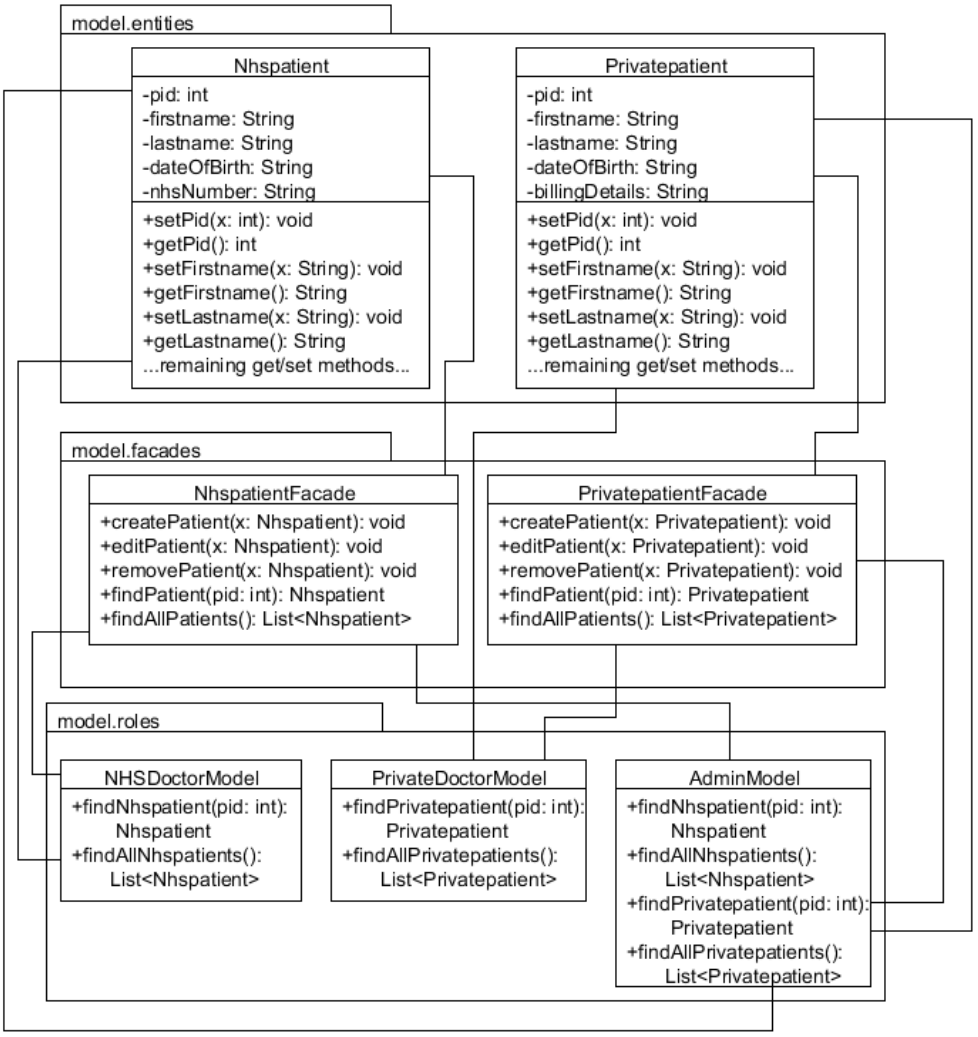

FIGURE 7.5: Example use of RBAC Model Pattern for GP Surgery Example. Resources are in package 'model.entites' and 'model.facades', role models in 'model.roles'.

7.6.2 RBAC Controller

Figure 7.6 illustrates the RBAC Controller pattern being used for designing the patient-related operations in the system. There are Role Controller classes for each of the roles discussed in the RBAC Model pattern example. The methods inside the controllers are responsible for processing different types of input from the views.

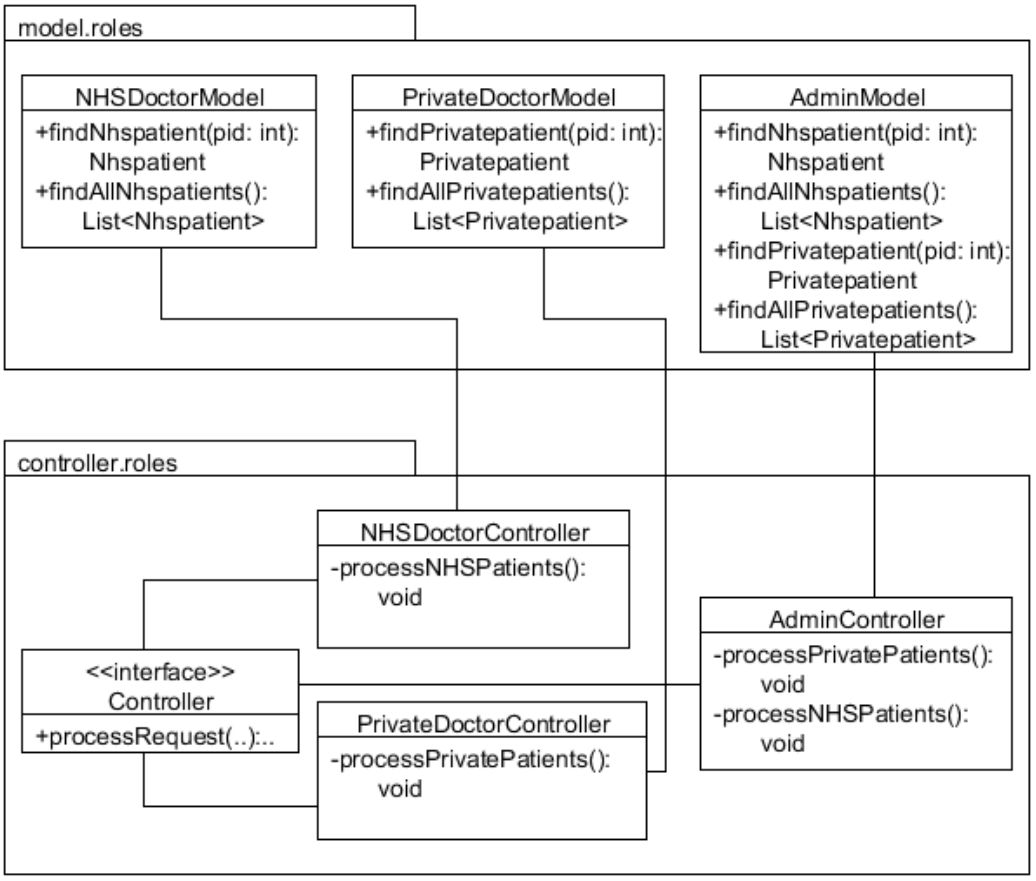

FIGURE 7.6: Example use of RBAC Controller Pattern for GP Surgery Example

7.6.3 RBAC View

Figure 7.7 illustrates the RBAC View pattern being used for designing the patient-related operations in the system. The view ending with 'ops' is for selecting operations on the type of patient specified in the name, and the view ending with 'details' is used to display that type of patient's data.

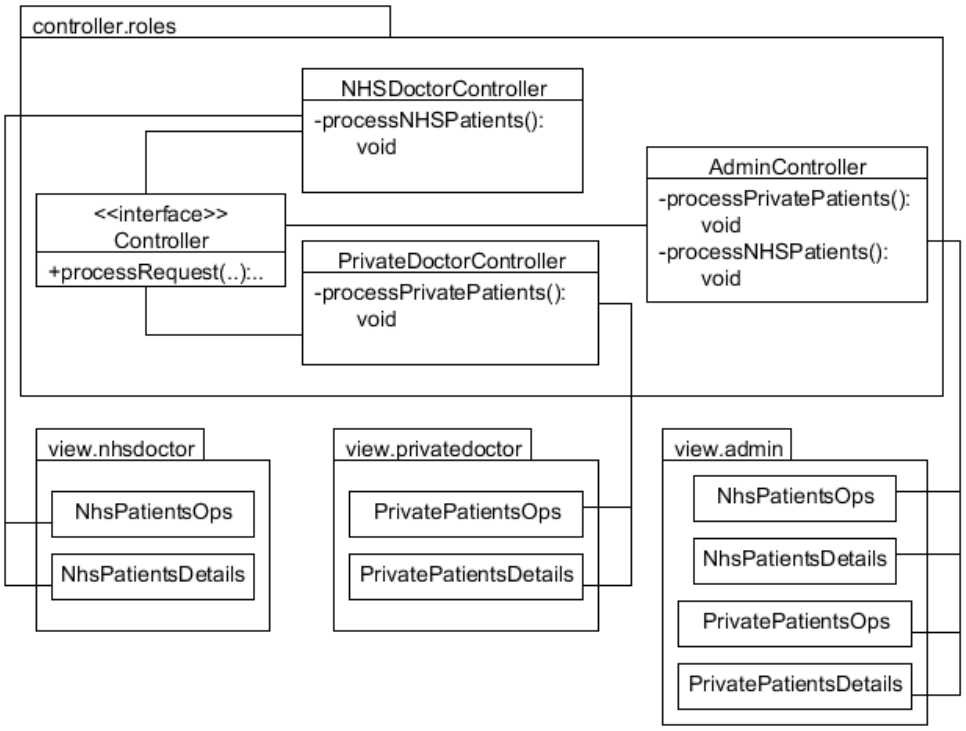

FIGURE 7.7: Example use of RBAC View Pattern for GP Surgery Example

Chapter 8

Static Enforcement Mechanism

Our source-level static verifier takes as input a well-formed program, which is defined as follows:

Definition 8.1 (Well-formed program)**.** A well-formed program consists of a (syntactically correct) JPol policy file and a Java program that implements the RBAC MVC patterns. Implementing the patterns means, in particular, the following. Firstly, inheritance is disabled between: resources, two role MVC classes belonging to different roles and a resource, a role MVC class, a session class and an 'other' class. Also, the Session classes: correctly authenticate users, activate the correct role(s) allowed for the user and switch roles correctly for the retrieved and selected roles.

A well-formed program might contain unauthorised calls to actions on resources. The static verifier should reject a program if an access violation is found, else accept it. In other words, it should only accept programs that satisfy the policy (see Definition 6.12). In this chapter, we describe high-level details of the algorithm for static checking. The first of two phases, described in Chapter 8.1, generates abstract syntax representations of the policy and program, and populates tables to be used in the second phase. The

second phase, described in Chapter 8.3, uses the abstract syntax tree and tables to check that the program satisfies the policy.

8.1 Parsing Policy and Target Program

The policy and program are parsed to produce their ASTs and generate relevant intermediate data structures needed for the static verifier. We have already described in Chapter 6.2 the AST and tables 'Resources' and 'Roles' generated by parsing the policy. In the rest of this section we describe the process of parsing the program.

8.2 Categorising Classes and Generating Data.

This first phase identifies the classes and categorises them as Resource classes, Role Model classes, Role Controller classes, Role View classes, Session Classes and Other classes (the latter contains any classes that do not fit into the other categories). This is illustrated in the top-level AST of the program in Figure 8.1.

The AST shown in Figure 8.1 is extended so that each class is represented by a tree under the node representing its category. The AST for all classes is the same regardless of category; parsing a class follows the standard parsing rules for Java classes. We show a simplified AST of a class in Figure 8.2.

In order to categorise each class, we use naming restrictions on the class names. The restrictions are described at a high level as follows. Note that 'class-id' is used as shorthand to represent any valid class identifier in Java and that 'Roles' and 'Resources' refer to the tables generated from the policy discussed in Chapter 6.2. Also note that 'Class' refers to the Class node in the AST currently being traversed. We use the notation $s_1 \ll s_2$ to specify that s_1 is a substring of s_2, $s_1 + s_2$ to specify that s_1 is

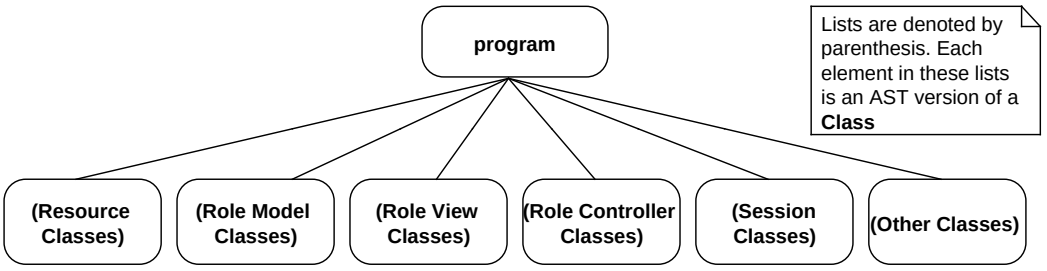

FIGURE 8.1: Top-Level Abstract Syntax of Program

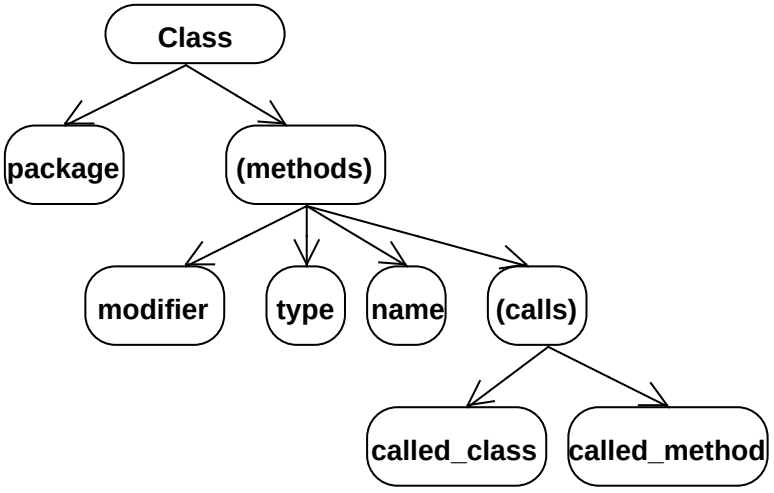

FIGURE 8.2: Abstract Syntax of a Class in the Program

concatenated with s_2, $tbl[n]$ to specify the n-th element of table tbl (where n is an integer).

- If the name of the class is the same as a resource, i.e.,

  $\exists n.\ 0 < n < Resources.size\ and\ Resources[n].name = Class.name$

  then it is a Resource Class.

- If the name of the class is the name of a role followed by the string 'Model', i.e.,

  $\exists n.\ 0 < n < Roles.size\ and\ Roles[n].name + `Model' = Class.name$

  then the class is a Role Model class.

- If the name of the class is the name of a role followed by the string 'Controller', i.e.,

  $\exists n.\ 0 < n < Roles.size\ and\ Roles[n].name + \text{`}Controller\text{'} = Class.name$

  then the class is a Role Controller class.

- if the name of the class is the name of a role followed by the String 'View', followed by any valid Java class identifier, i.e.,

  $\exists n.\ 0 < n < Roles.size\ and\ Roles[n].name + \text{`}View\text{`} + \text{`class-id'} = Class.name$

  then the class is Role View class.

- if the name of the class begins with the string 'Session', i.e.,

  $\text{`}Session\text{'} + \text{`class-id'} = Class.name$

  then the class is a Session Class

After identifying the category of all the classes, this phase also generates tables containing the names of all classes in each category except the category of 'Other classes'. We call these tables 'ResourceClasses', 'RoleModelClasses', 'RoleControllerClasses', 'RoleViewClasses' and 'SessionClasses'. This is to simplify the process of looking up called classes in the checks made by the verifier (discussed below).

8.3 Static Checks

The second phase performs checks of two kinds: checking the modifier of a declared method and checking the class that a method is invoked on. Checking the modifier of a method requires extracting the value of 'modifier' in each element of the list '(methods)' from the abstract syntax tree (AST) for the class (see Figure 8.2 for the abstract representation of a class). The modifier's correct value will depend on the

details of the check being done. Checking the class that a method is invoked on requires extracting the value of 'called_method' when checking each element of the list '(calls)', when checking each element of the list '(methods)' in a class. The call's validity will be determined based on the kind of check being done. Both kinds of checks are discussed below. They are performed by traversing the AST of the program starting from the root. For each category of class (Resource, Role Model, Role Controller, or Other class), the algorithm performs specific checks as follows.

8.3.1 Resource Class Checks.

The checks on Resource classes, performed by a subprogram of the verifier, called ResourceClassChecks, are described below.

1. For each method (i.e., each element of the list '(methods)', see Figure 8.2), we search the actions sub-table (generated when parsing the policy, see Figures 6.1 and 6.2) for 'Class.name' (which is the name of a resource) then:
   (a) If the method name is in this sub-table, then the value of the node 'modifier' must be 'public'.
   (b) Else the value of the node 'modifier' must be 'private'.
2. For each call (each element of the list '(calls)', see Figure 8.2), we check that:
   (a) The called class (the node 'called_class') is not the name of a Role Model class. This is done by searching the names of classes in the table 'RoleModelClasses'.
   (b) The called class is not the name of a Role Controller class. This is done by searching the names of classes in the table 'RoleControllerClasses'.

(c) The called class is not the name of a Role View class. This is done by searching the names of classes in the table 'RoleViewClasses'.

(d) The called class is not the name of a Session class.

Summarising: Resource classes cannot contain invocations to methods in Role Model, Role View, Role Controller, or Session classes.

8.3.2 Role Model Class Checks.

The checks on Role Model classes, performed by a subprogram called RoleModelClass-Checks, are described below.

1. First obtain the name of the associated Role for this class by removing the sub-string 'Model' from 'Class.name'.

2. For each call, we check that:

   (a) If the called class is a Resource class, then

      i. If the called method (the node called_method) is an action, which is done by searching the 'actions' sub-table for that resource in the table 'Resources' generated when parsing the policy, see Figures 6.1 and 6.2, then the pair of values ['called_class', 'called_method'] must appear in the permissions for the associated Role of the class (done by searching the 'permissions' sub-table of the matching role in table 'Roles')

   (b) The called class is not the name of a Role Controller class.

   (c) The called class is not the name of a Role View class.

   (d) The called class is not the name of a different Role Model class. This is done by checking if the called class contains the substring 'Model', the name of the class must be the same as the value in the node 'Class.name'.

(e) The called class is not the name of a Session class.

Summarising: Role Model classes cannot contain invocations to methods in Role View, Role Controller or Session classes; in addition, they cannot contain invocations to methods in a Role Model class associated to a different role, and they can only contain invocations to actions on resources if authorised by the policy for their role.

8.3.3 Role Controller Class Checks.

The checks on Role Controller classes, performed by a subprogram called RoleControllerClassChecks, are described below.

1. First obtain the name of the associated Role for this class by removing the substring 'Controller' from 'Class.name'.
2. For each call we check that:
   (a) If the called class is a Resource class, then
      i. If the called method is an action the pair of values ['called_class', 'called_method'] must appear in the permissions for the associated Role of the class.
   (b) The called class is not the name of a different role's Role Model class. This is done by checking if the value of 'called_class' contains the substring 'Model', then remove this substring and check that it matches the Role of this class.
   (c) The called class is not the name of a different role's Role Controller class. This is done by checking if the value of 'called_class' contains the substring 'Controller', then the called class must be the same as the value in 'Class.name'.

(d) The called class is not the name of a different role's Role View class. This is by checking if the value of 'called_class' contains the substring 'View', if so then remove this substring and the following characters in the name up to the end, and check that it matches the role name of this class.

(e) The called class is not the name of a Session class.

Summarising: Role Controller classes cannot contain invocations to methods in Session classes; in addition, they cannot contain invocations to methods in a Role Model, Role View or Role Controller class associated to a different role, and they can only contain invocations to actions on resources if authorised by the policy for their role.

8.3.4 Role View Class Checks.

The checks on Role View classes, performed by a subprogram called RoleViewClass-Checks, are described below.

1. First obtain the name of the associated Role for this class by removing the substring 'View' and the following characters up to the end from 'Class.name'.

2. For each call we check that:

   (a) If the called class is a Resource class, then

      i. If the called method is an action the pair of values ['called_class', 'called_method'] must appear in the permissions for the Role of the current class.

   (b) The called class (i.e., the node 'called_class') is not the name of a Role Model class.

   (c) The called class is not the name of a different role's Role Controller class.

(d) The called class is not the name of a different role's Role View class.

(e) The called class is not the name of a Session class.

Summarising: Role View classes cannot contain invocations to methods in Role Model or Session classes; in addition, they cannot contain invocations to methods in a Role View or Role Controller class associated to a different role, and they can only contain invocations to actions on resources if authorised by the policy for their role.

8.3.5 Session Class Checks.

The checks performed by the static verifier on Session classes are specified in a subprogram called SessionClassChecks. For each call, we check that:

1. The called class is not a Resource Class.
2. The called class is not a Role Model Class.

8.3.6 Other Class Checks.

The checks performed by the static verifier on Other classes, using the tables 'ResourceClasses', 'RoleModelClasses', 'RoleControllerClasses' and 'RoleViewClasses', are as follows. They are specified in a subprogram called OtherClassChecks.

1. No invocation of a Resource Class method.
2. No invocation of a Role Model Class method.
3. No invocation of a Role View Class method.
4. No invocation of a Role Controller class method.
5. No invocation of a Session class method.

8.4 Properties

The static verification algorithm described in the previous sections ensures that the programs that pass the checks do not perform invalid access requests. More precisely, programs satisfy the following properties (note that at this stage of the work, proofs are intuitive explanations instead of formal proofs):

Definition 8.2 (OK program)**.** A program P is OK, written $OK(P)$, if:

1. Its actions are 'public' and auxiliary methods are 'private',
2. Resource classes do not invoke methods of a Role Model, Role Controller, Role View or Session class,
3. Role Model methods do not invoke methods in Session, Role Controller or Role View classes or an action that is not allowed by the policy for the associated Role,
4. Role Controller classes do not invoke Session Classes or an action that is not allowed by the policy for the associated Role,
5. Role View methods do not invoke methods in Role Model or Session classes or an action that is not allowed by the policy for the role that the Role View class belongs to,
6. Role classes do not call classes belonging to other Roles;
7. Session Classes do not invoke Role Model classes and do not invoke methods in Resource classes.
8. Other Class methods do not invoke a method of a Resource, Role Model, Role Controller, Role View, or Session class.

Theorem 8.3. *If a well-formed program P is accepted by the static verifier, then $OK(P)$.*

Proof. The algorithm traverses the abstract syntax tree of the program, and for each kind of class, the subalgorithms ResourceClassChecks, RoleModelClassChecks, RoleControllerClassChecks, RoleViewClassChecks, SessionClassChecks and OtherClassChecks ensure that the property holds. We consider each part in Definition 8.2 in turn.

Part 1 is a consequence of the checks performed in the ResourceClassChecks algorithm (see Chapter 8.3.1), specifically checks 1(a) and 1(b) where the program is rejected if the modifiers for methods in the Resource classes are not 'public' for actions and 'private' for auxiliary methods; in check 2, all the invocations within resource class methods are checked to make sure that they do not call on a method from a Role Model, Role Controller, Role View or Session class, as required in part 2.

Similarly, parts 3, 4, 5 and 6 are a consequence of the checks performed in the subalgorithm RoleModelClassChecks, RoleControllerClassChecks and RoleViewClassChecks, respectively.

Part 7 is a consequence of SessionClassChecks checks 1 and 2, which reject a program if in a Session class method there is an invocation of a Resource class or Role Model class.

Part 8 is a consequence of OtherClassChecks checks 1, 2, 3, 4 and 5 respectively, which reject a program if an Other class method contains a call to a method in a Resource, Role Model Role Controller, Role View or Session class, respectively. □

Theorem 8.4. *If a well-formed program P is rejected by our verifier, then not $OK(P)$.*

Proof. If the program has been successfully parsed but the verifier rejected it, it is because one of the algorithms ResourceClassChecks, RoleModelClassChecks, RoleControllerClassChecks, RoleViewClassChecks and OtherClassChecks detected an error. If the program is rejected by the algorithm ResourceClassChecks, it is because a method

in a Resource class has an incorrect modifier as a result of check 1, or because there is an invocation to a forbidden method as a result of check 2. This could be a method in a Role Model, Role Controller, Role View class or a Session class.

If the program is rejected by the algorithm RoleModelClassChecks, it is because a method in a Role Model class contains an invocation of an action which is not permitted by the policy for role that the Role Model belongs to, as a result of check 2(a), or it contains another forbidden invocation. This could be a method in a Role Controller or Role View class, as an outcome of checks 2(b) and (c) respectively, or a method of a Role Model class belonging to a different role, as a consequence of check 2(d) or a method of a Session class (check 2(e)).

If the program is rejected by the algorithm RoleControllerClassChecks, it is because there is a call to a forbidden method. This could be an action that is not permitted by the policy, due to check 2(a), or a method in a Role Model, Role Controller or Role View class that belongs to a different role, resulting from checks 2(b), (c) or (d) respectively, or a method of a Session class (check 2(e)).

If the program is rejected by the algorithm RoleViewClassChecks, it is because there is a method in a Role View class which contains a call to an action that is not permitted by the policy, resulting from check 2(a), or there is another forbidden method invocation. This could be a method in a Role Model class, due to check 2(b), or a method in a Role Controller or Role View class that belongs to a different role, the consequence of checks 2(c) and (d) respectively, or a method of a Session class (check 2(e)).

If the program is rejected by the algorithm SessionClassChecks, it is because there is a method in a Session class which contains a call to a forbidden method. This could be a call to a method in a Resource or Role Model class, resulting from checks 1 and 2 respectively.

If the program is rejected by the algorithm OtherClassChecks, it is because there is a method in an Other class which contains a call to a forbidden method. This could be a call to a method in a Resource, Role Model, Role Controller or Role View class or a Session class, resulting from checks 1, 2, 3, 4, or 5, respectively. □

Theorem 8.5. *A well-formed program P accepted by the verifier satisfies the policy (see Definition 6.12).*

Proof. To prove that P satisfies the policy according to Definition 6.12 we need to show that only authorised users with active role r having permission $[Res, m]$ can invoke the action m of an instance of Res. Let $res.m$ be a call to m in the program P, for which the parser has identified the called class to be Res and the called method to be an action m. Since P is *well-formed*, by Definition 8.1 it implements the RBAC MVC patterns. Then, a user u can only execute $res.m$ if the user has been authenticated and is in a session, where by Definition 6.10, one of u's roles, say r, has been activated.

By Definition 6.11, this implies that r's Role Controller has been invoked. Moreover, since P has been accepted by the verifier, by Theorem 8.3, $OK(P)$. Once the Role Controller for r has been invoked, by Definition 8.2 the Java code executed from the role classes associated to r contains only invocations to actions $res.m$ that are authorised by the policy, and there are no calls to methods in session classes or methods in role classes belonging to a different role. Moreover, any called method in a class which is not one of r's role class will not contain an invocation to an action (unless it is a method in a resource class) or to a role class (by Definition 8.2). Therefore once the Role Controller for r has been invoked, all the called methods are authorised. Note that the only classes outside role classes which could call a role class are session classes, which, since the program is well-formed, must satisfy the requirements of the RBAC-MVC pattern. In particular, we trust the calls to Role Classes made in Session Classes.

□

To provide flexibility to programmers, we have allowed actions to be invoked within Resource classes. However, this means that there may be an indirect violation of the policy, e.g. role n invokes action a which is allowed, which in turn invokes action b, which is not allowed, for n in the policy. Currently, we assume that indirect calls are not a policy violation (i.e., the policy specifies the actions that a role is allowed to call, and it does not restrict the invocations within those actions). The Session classes are the critical part of the program in our approach, in which Role class invocations are trusted and not verified. The minimal Trusted Computing Base in our approach is therefore the action methods and the Session classes.

In future work, we will extend the verifier to include checks within actions, to alert programmers if there is an indirect call to an action not allowed by the policy. Our verifier could alert programmers by giving warning messages if this happens.

8.5 Implementing the Static Verifier

Our implementation consists of a JPol policy parser, produced using the ANTLRWorks tool [13], and a static analysis program which are both part of a plug-in we have produced for the Eclipse Integrated Development Environment (IDE) [19]. Further discussion on the justification of these choices will follow in Part IV.

We have tested our plug-in on a simple doctor's surgery web database application implemented in Java Enterprise Edition (JEE) (refer to [27] for an overview of JEE). The tool outputs helpful error messages in Eclipse's editor window, consisting of the class name and line number where the error occurs, the kind of error that has occurred (e.g.

'Invocation not permitted') and a description of why that error could have occurred. Refer to the next Chapter and to Chapter 15 for details.

Chapter 9

Case Study: GP Surgery System

We have implemented the GP surgery running example (see Chapter 1.2.1) as a web database application in Java. The data is stored in a database using Apache Derby and the application is built in JEE using Eclipse Integrated Development Environment (IDE) using the Oracle Glassfish application server. The structure of the application follows from the UML diagrams in Figures 7.5, 7.6 and 7.7. We explain the design and implementation, further, below.

9.1 Server-Side Components

The structure of the components to be held on the server follows the RBAC Model pattern. Therefore, the server contains resource classes, which in the case of the GP surgery are Java Persistence API (JPA) entity classes that represent a row in one table in the underlying database, and Session EJB which perform the business logic of the application. The structure of the application is shown in the UML diagrams in Chapter 7.6. Session EJBs suffixed with the string 'Facade' are responsible for performing database queries for the table that they share their name with. The methods in these

Session EJB Facades may use or return instances of the JPA entity class for that table. JPA entities and Session EJB Facades have been stored in the packages 'model.jpa' and 'model.ejb' respectively, for convenience. The server also contains the Role Model classes, which are stored in the package 'model.roles' (although there is no requirement that Role Model classes must be stored in their own package or separately from resource classes).

There are two extra Session EJB classes, which both fall outside of the scope of the specification and enforcement of the RBAC policy in our approach. The first of these is 'LoginBean', which is used to check the user's details to authenticate and 'log in' the user. We do not restrict the method for authentication therefore such an EJB would not be expressed as a resource in the policy, thus we do not control its implementation or usage. The second is 'DrugListBean', which is a simple class used to store data retrieved from the user when they enter a list of entries to put in the table 'Nhsprescriptiondrugs' or 'Privateprescriptiondrugs'. Each entry in the list will contain the ID of the Drug to be prescribed, the quantity to be prescribed and the dosage. The user will enter multiple sets of these values into the user interface (UI), and in order to store them, each set will be an instance of the DrugListBean and the set as a whole is stores as a List in Java. The data inside this EJB is not confidential, since no information on who the prescription is intended for or created by is stored, therefore it does not need to be expressed as a resource.

In the server-side components, some task methods in the Role Model classes have just one line, to call a method from a Resource class. This means that to program the application according to the design pattern proposed, in some cases extra lines of code are needed to 'wrap' a call. However, in our experience few methods needed this, meaning that the overall code additions were very small.

9.2 Client-Side Components: Application Client

For the application client, we have implemented role controllers and role views as plain Java classes. Role controllers are stored in package 'controller' and role views in multiple packages named 'view.rolename' (where 'rolename' is replaced by the name of the role to which the role views will belong). Both of these are again for convenience. The role controllers contain data fields which are instances of the role model belonging to the same role to which the controller belongs. When the role controller is instantiated, it retrieves an instance of the role model from the server via a 'lookup' (more details of this process can be found in [27]).

The role views interact with role controllers by passing string parameters to the controller, which represent inputs such as chosen operations to execute. For example clicking a button labelled 'viewAllNHSPatients' will pass this string to the controller. The role controller checks the received parameters from the views, and decides what action to take, which could be to invoke a task method contained in the role model class. The role controller will usually then pass some data (i.e. object(s)) to one of the views and present it to the user. The chosen view would then read any received data and present its information to the user along with any other behaviour that it executes. An example of this flow is the following. The user may click on a button in the the running instance of an 'NHSDoctorView' class, which passes the string parameter 'viewAllNHSPatients' to the instance of the 'NHSDoctorController' class which it holds as a data field. This then checks the value of the parameter and will invoke the task method 'findAllNHSPatients()' on the instance of the 'NHSDoctorModel' class that it holds as a data field. This method has the return type 'List<Nhspatient>' which is a Java List of instances of the JPA entity class 'Nhspatient' that represents a row from the 'Nhspatients' table. The Role Controller passes this List to the 'AdminViewNHSPatientList' Role View, which reads each 'Nhspatient' object in the list

to write its details in a table by calling its attributes e.g. ‘firstname’, ‘lastname’, etc.

9.3 Policy

In a client-server scenario in Java, resources exist as server-side EJB classes whose functionality is shared by several clients - called by Remote Method Invocation (RMI). The policy must do the following:

1. declare each resource,
2. add to each resource each of its actions,
3. declare each role and
4. add to each role the list of permissions it has to invoke actions on resources.

Listings 9.1 and 9.2 show an extract of a sample policy for the GP surgery example for Resources and Roles, respectively.

```
1  // (1) Declare Resource ``Nhspatient"
2  Resource nhspatient = new Resource(``Nhspatient");
3  // (2) Add each of its Actions
4  nhspatient.addAction(``Nhspatient");
5  nhspatient.addAction(``setFirstname");
6  nhspatient.addAction(``getFirstname");
7  nhspatient.addAction(``setLastname");
8  nhspatient.addAction(``getLastname");
9  nhspatient.addAction(``setDob");
10 nhspatient.addAction(``getDob");
11 nhspatient.addAction(``setNhsnumber");
12 nhspatient.addAction(``getNhsnumber");
13 nhspatient.addAction(``setPid");
```

```
14 nhspatient.addAction(``getPid");
15 // ...add more actions...
16 // Do (1) and (2) for each Resource
```

LISTING 9.1: Extract of Sample JPol Policy declaring Resources with their actions

```
17 // (3) Declare Role ``NHSDoctor" and
18 Role nhsdoctor = new Role(``NHSDoctor");
19 // (4) Add each its Permissions
20 nhsdoctor.addPermission(nhspatient, ``getFirstname");
21 nhsdoctor.addPermission(nhspatient, ``setFirstname");
22 nhsdoctor.addPermission(nhspatient, ``getLastname");
23 nhsdoctor.addPermission(nhspatient, ``setLastname");
24 // ...add more Permissions...
25 // (3) Declare Role PrivateDoctor
26 Role privatedoctor = new Role(``PrivateDoctor");
27 // (4) Add each of its permissions
28 // Note that it has no permissions on
29 // Resource ``Nhspatient"
30 // Do (3) and (4) for each Role
```

LISTING 9.2: Extract of Sample JPol Policy declaring Roles with their permissions

From the extracts, we can see some key actions of a resource 'Nhspatient' that have been declared. These correspond to the the design of the class 'Nhspatient' in the UML diagram in fig 7.5. The Resource classes and their associated actions must all be declared, because the verifier will check that these classes have implemented the actions as per the policy. We can also see two Roles have been declared. Firstly, 'NHSDoctor' which we show has permissions on (all of the actions of) the Resource 'Nhspatient'. Secondly, 'PrivateDoctor' which we show has no permissions on the Resource 'Nhspatient'.

9.4 Applying Verification

Let us first discuss the case where the program passes the checks. If there are no errors found by our verifier, then, according to the policy extracts, classes belonging to 'PrivateDoctor' will not have any calls to methods in 'Nhspatient' class, but 'NHSDoctor' may. Now, we can modify the program to show, from these small extracts of the policy, examples of the kinds of errors that can be caught at compile-time by our static verifier.

9.4.1 Undefined Action.

In this type of error, a public method has been implemented in a resource class in the program but not defined as an action in the policy. For example, in Listing 9.1 if we remove line 5, the action 'setFirstName' would be undefined. When our verifier checks the program, it would read a public method named `'setFirstName'` in the 'Nhspatient' class. Since it is a method of a resource, it must either be a private method - auxiliary - or must exist as an action in the policy for the resource 'Nhspatient', else the program will be rejected. This prevents public methods of resources from being ignored from the policy and program validation process; the main objective of our access control approach is to restrict invocation of visible methods of resource classes. Figure 9.1 shows this error being caught and the resulting error message.

9.4.2 Invocation Not Permitted.

In this type of error, an action of a resource is called in a class belonging to a role which is not permitted for that role according to the policy. For example, we restore line 5 in Listing 9.1, then in the class 'PrivateDoctorModel', we invoke the action 'setFirstName' on the Resource 'Nhspatient'. We did not assign any permissions on

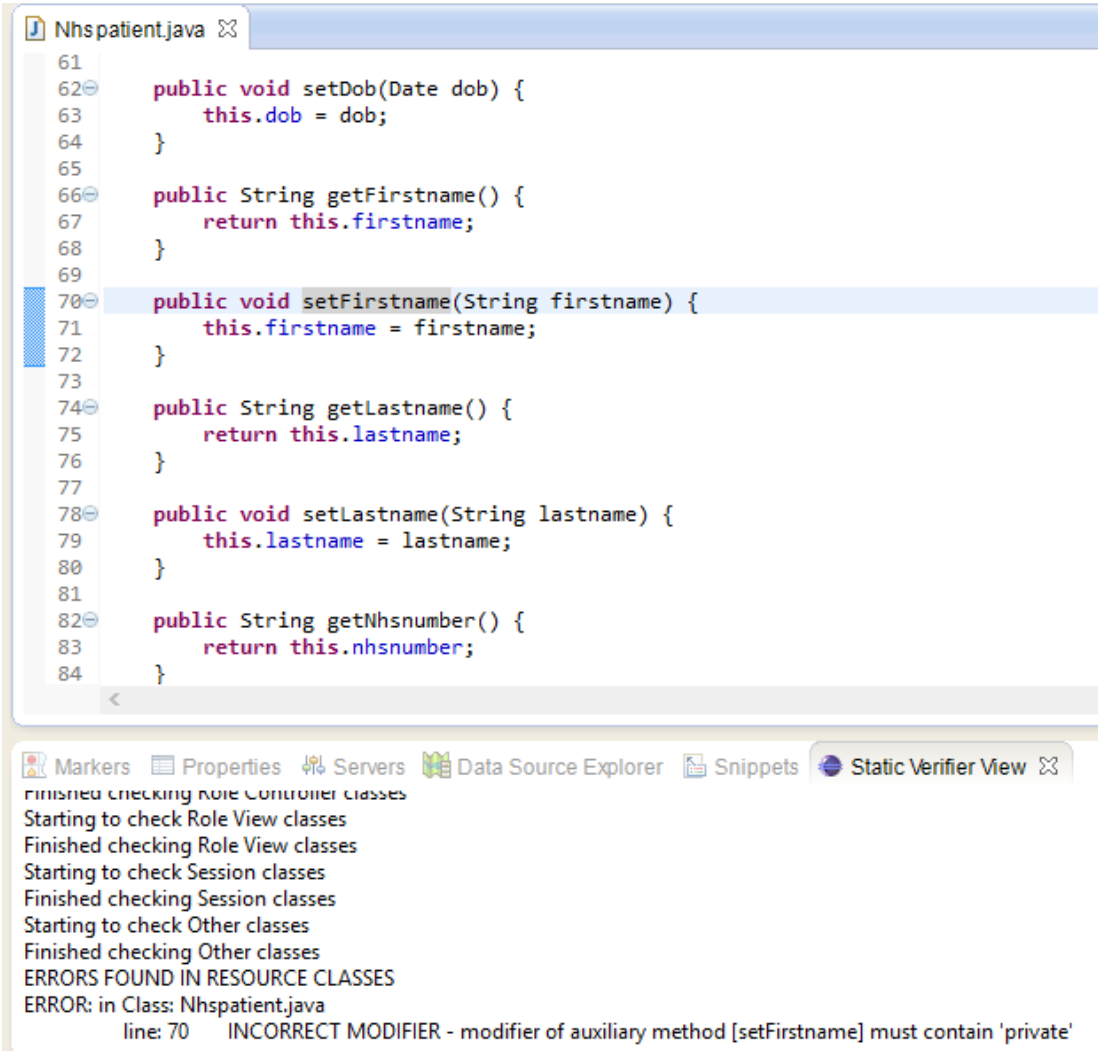

FIGURE 9.1: Example of Undefined Action error

'Nhspatient' to Role 'PrivateDoctor', and our verifier reads the invalid invocation and rejects the program. The error and resulting error message is shown in Figure 9.2.

9.4.3 Invocation Between Roles.

In this type of error, a class belonging to role x calls a method in a class belonging to a role y, where $x \neq y$. For example, role model class 'PrivateDoctorModel' containing an

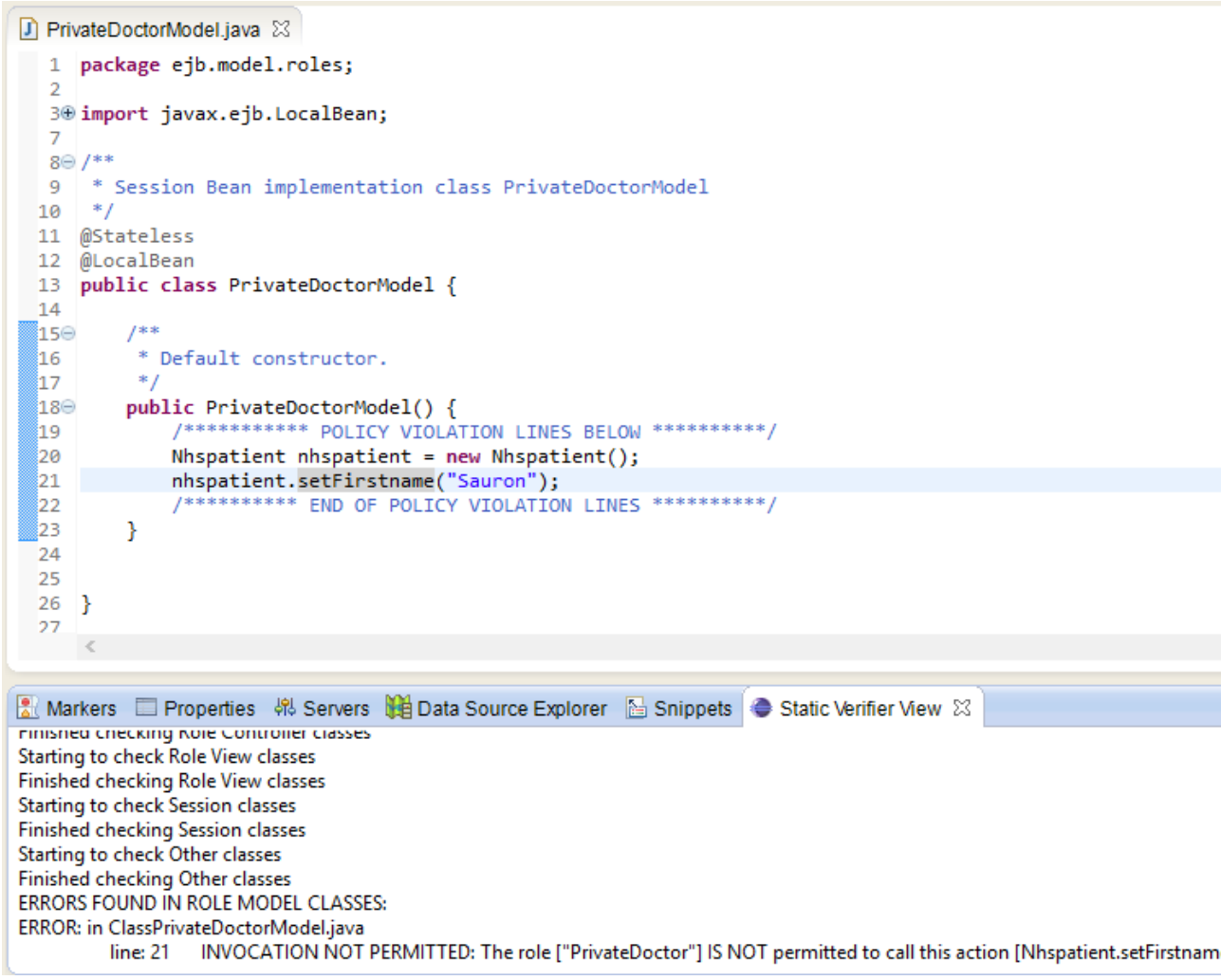

FIGURE 9.2: Example of Invocation Not Permitted error

invocation to a method 'createNHSPrescription' of the 'NHSDoctor' role model class. Figure 9.3 shows this error and the resulting error message when the verifier detects it.

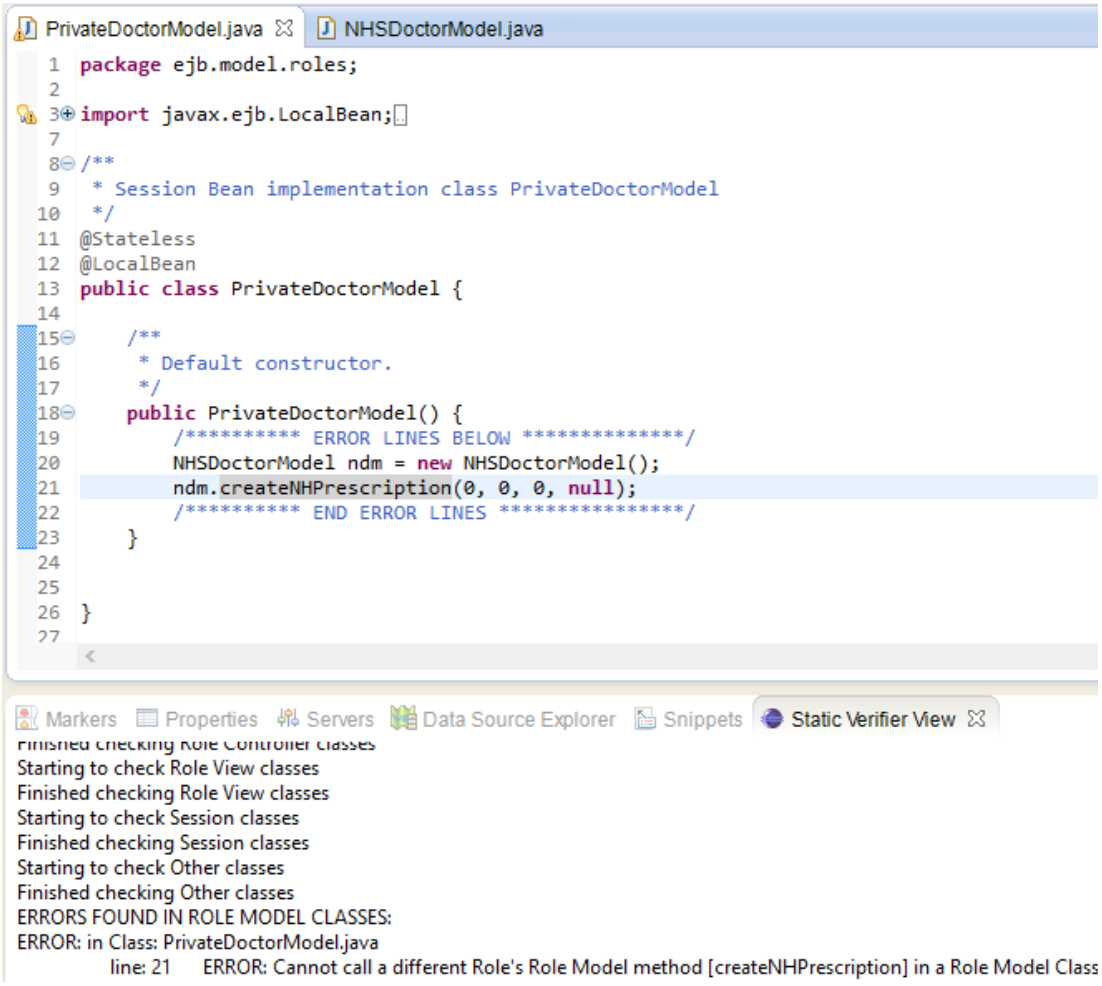

FIGURE 9.3: Example of Invocation Between Roles error

Part III

Hybrid Enforcement of CBAC

Chapter 10

Concept Overview for Hybrid Enforcement

Our second system extends the first to incorporate dynamic policies and enforcement in conjunction with the static policies and enforcement, thus producing a hybrid approach. We target policies following the CBAC model, which is a general meta-model of access control in which dynamic policies can be specified. One of the key notions in CBAC is the classification to which users are assigned, which is known as a *category* (in the first system, which targeted RBAC, this was a *role*). Categories are assigned permissions to invoke actions on resources.

Then, users are assigned to categories, and may be assigned to multiple categories. This user-to-category assignment is where the dynamic features of categories can come into effect. It is possible for users to be assigned to a category with some condition that utilises dynamic information i.e. situational or event information as specified in CBAC. For example, users may be assigned to a particular category based on their location, which would be an example of a condition utilising situational information.

The static parts of a CBAC policy need to be separated and enforced statically, whereas the dynamic parts need to be enforced dynamically. We describe our approach to solving this problem by first adapting the concept overview and our static approach (including the patterns and static enforcement mechanism) to a CBAC context, then proposing additional patterns and a code generation phase after the static enforcement phase, where the generated code implements the dynamic enforcement as an in-lined reference monitor.

We now describe a high-level overview of the approach, beginning with policy concepts then program concepts. Note that we reuse several concepts from the previous static system. In our approach, the access restrictions are specified using CBAC. Resources, their action and auxiliary methods and access requests are all exactly as defined in the last system (see Chapter 6). Permissions are similar except for the fact that they are now assigned to categories, instead of roles, which are defined as follows. Note that it is user-to-category assignment that can use conditions utilising situational or event information, not permission-to-category assignment.

Definition 10.1 (Category)**.** A *category* is a distinct group, to which users and permissions are assigned. At the policy level, a *category* consists of a name and a list of permissions to access resources. User membership to a *dynamic category* depends on conditions using dynamic information, but does not for membership to a *static category*. Categories can be arranged in a hierarchy, where a senior category *subsumes* a subordinate - the permissions of the subordinate are copied to the senior. Lastly, we define a relation *can-be*, written $c_1 \rightarrow c_2$, which states that a user that is in category c_1 can, under some conditions, transfer to **dynamic** category c_2. Two categories in a *can-be* relation do not have to have any common permissions. Note that subsuming implicitly means that the subordinate category *can-be* the senior (and vice-versa). We also refer to categories in a can-be relation as *related* categories.

An informal example of a *can-be* relation is: *Employee* → *EmergencyEmployee*, where the former can transfer to the latter in an emergency. The two categories can have distinct permissions i.e. only the first can use the lifts.

Moving on to the concepts of the target program, there is a problem that the policy is concerned with restricting actions in resources, but users of a target program on which a policy is enforced do not interact with the program by seeking to invoke actions - programs abstract these lower level details away from the user. In many cases, users do not know what the resources and actions are.

To link the user's view of a system with the policy, we define that users interact with the system to perform a *task* (see Definition 10.2), which may or may not access resources. Utilising this concept, we present the general flow of a program implementing a CBAC policy (or one of its specialisations) compared with the flow in our approach, see Figure 10.1. The general flow begins with authentication (i.e. login), then reaching a phase we call *Tasks*, in which the user can perform tasks in the system. Some of these may access resources, and so every access request is intercepted by a reference monitor, which allows the request to continue only if the user is computed to have the required category (which has the permission to invoke that request). In our approach, after logging-in, users enter a *Session Interface*, from which they can then access four interfaces depending on the four kinds of task they wish to perform in the system. A *task* is similar to its definition in the previous system, however role tasks are replaced as described below.

Definition 10.2 (Task)**.** Role tasks are replaced with the following two kinds of tasks. Firstly, a *Static Category Task* is an operation to be performed by an authorised user in a specific Static Category, possibly invoking one or more actions. Secondly, a *Dynamic Category Task* is similar to a Static Category task, except is to be carried about by an authenticated user in a specific Dynamic Category.

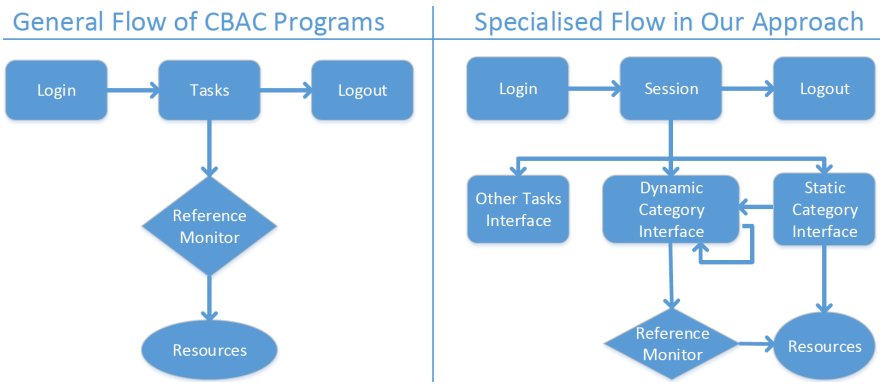

FIGURE 10.1: General and specialised flow of programs that enforce CBAC.

Static/Dynamic category tasks are handled by static/dynamic category interfaces, which are similar to role interfaces, except each kind of category (instead of a role) is implemented as a set of MVC components. The naming restrictions used to identify each kind of category MVC class is also very similar to the first system, except that the name of each kind of class must begin with the name of a category (instead of a role).

In the previous system, the relation between roles was through hierarchies, which were not actually implemented in the program. Each role's set of MVC components were kept completely separate and invocations from one role's MVC components to another role's was disallowed by the static verifier. Instead, the policy language parser copied the permissions of a subordinate role to the more senior one.

However, categories contain additional relations, which are to be implemented in the program in our approach. To explain the link between the category interfaces as shown in Figure 10.1, we first define the concept of a *link* between two category interfaces.

Definition 10.3 (Link)**.** Two category interfaces are *linked*, written $c_1 \triangleright c_2$, if there is an invocation of the category interface of category c_2 in the category interface of category c_1 to enable the user to move from one category interface to another. It is

implemented as a call to any method of the category controller of c_2 in the category model of c_1.

Firstly, for each category s, and the set of its *can-be* related dynamic categories, the following condition must hold: $\forall x(s \rightarrow x \Leftarrow s \triangleright x)$. This way, when the user interacts with the program in a static or dynamic category's interface, they can then invoke the interfaces of *can-be* related dynamic categories.

Session tasks are handled by the *Session Interface*. This is also similar to its definition in the previous system. However, the session must now retrieve the authenticated user's policy-assigned static categories, and activate the selected static category (instead of role). Also, each session holds and maintains a single *security context*, which is a set of security-relevant data which will be needed for dynamic checks e.g. the user's username, the date/time, e.t.c.

As before for the session, the authentication process is left open to the programmer so that heavy restrictions are avoided. The notion of authenticated user is similar to our first system with the notion of a role replaced by the notion of a static category. This is also true for static category activation.

The user can always switch between the category interfaces of any of their static categories. Once in a category interface, from it the user can access the category interfaces of the dynamic categories that are *can-be* related to that category (due to Definition 10.3). This provides a way for the user to interact with the program by exploring, from one category interface to the next, until they find the task they desire to execute.

At compile-time, we check that each action call in every class in a static category interface abides by the permission-to-category assignments in the policy. This can be done because of our design patterns which guide the implementation of the above described program flow. This flow enables us to know for each action call in a class in a category

interface, which category is calling that action - that category interface's owner. For static category interfaces, this means that no run-time check is necessary, because its action calls will have been checked with the permission-to-category assignments in the policy and the user-to-category permissions do not depend on any dynamic information - the user is always a member of an assigned static category. For dynamic category interfaces, we also need to check that at run-time, each action call abides by the user-to-category assignments in the policy i.e. the user is computed to be in a category which contains the permission to invoke that action. This is done by generating code before each action call, for a call to a programmer-provided reference monitor class which we call a *categoriser*. These contain the code to compute if a user is in one of the categories which contain the permission to call a specified action. We leave the implementation of the categoriser open to the programmer; reference monitors are well-studied in the literature (a classification of enforcement approaches, including dynamic and static, is presented in [20]).

We also leave open the implementation of session classes, in which we trust the calls made to classes in category interfaces and that the static categories retrieved for a user after authentication are correct according to the policy.

Chapter 11

Policy Language for Hybrid Enforcement

We adapt our previous policy specification language to CBAC and rename it to *JPolCat*. In this language, the syntax is slightly modified to make it more compact. Resources together with their associated lists of actions can be specified as they could be before, however instead of roles, categories can be specified together with their associated permissions, hierarchic and *can-be* relations. Static and dynamic categories are declared separately. These are all the static parts of a CBAC policy. User definitions are omitted, as in JPol, to provide some flexibility. The dynamic part - the user-to-category assignments - are not specified in JPolCat for two reasons. Firstly because we do not deal with authentication in our approach and secondly because of the generality required by CBAC (the dynamic information used in the conditions could be anything). Thus, the programmer will have to implement these assignments themselves, which is discussed further in the following chapter. As in JPol, we assume that only the access requests that are allowed are expressed, so all other requests are not allowed.

11.1 Syntax and Representation

The policy language adopts a compact, simple and efficient syntax. However, the static algorithm relies solely on the information generated as a result of parsing the policy file. Thus, the syntax can change and be adapted to any environment using CBAC.

The grammar of the JPolCat is as follows. The abstract syntax of JPolCat is illustrated in Figure 11.1.

```
17 stmts    ::=     (stmt END)+;
18 stmt     ::=     (sCat | sCatSubsume
19          | dCat | dCatSub | decRes
20          | canBe | COMMENT);
21 decRes   ::=     'Resource' ID '=' '[' acts;
22 acts     ::=     STRING ',' acts | STRING ']';
23 sCat     ::=     'Category' ID '=' '[' perms ']';
24 sCatSub  ::=     'Category' ID 'subsumes' '[' multid
25          | 'Category' ID 'subsumes' '[' multid '=' '[' perms;
26 dCat     ::=     'Category*' ID '=' '[' perms ']';
27 dCatSub  ::=     'Category' ID 'subsumes' '[' multid
28          | 'Category*' ID 'subsumes' '[' multid '=' '[' perms;
29 multid   ::=     ID ',' multid | ID ']';
30 perms    ::=     '(' ID ',' '[' acts ')' ',' perms
31          | '(' ID ',' '[' acts ')' ']';
32 canBe    ::=     ID 'can-be' '[' multid;
```

Listing 11.1: Extract of Sample JPol Policy declaring Roles with their permissions

The AST produced by the parser of JPolCat is shown in Figure 11.1. The generated tables are called 'Resources', 'SCategories' (denoting static categories), 'DCategories' (denoting dynamic categories) and 'CanBe' (showing each individual can-be relation between two categories as a row in the table) - containing the information needed for the static verification.

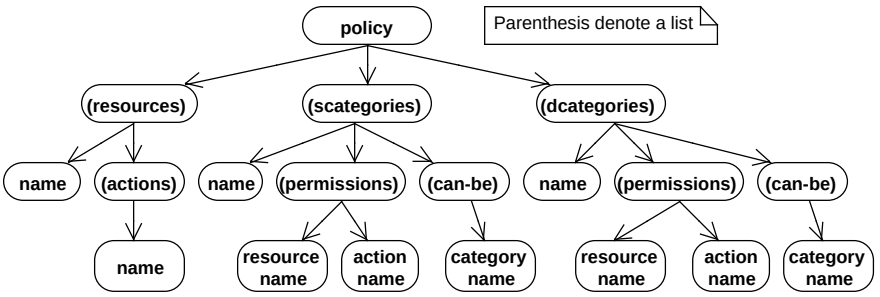

FIGURE 11.1: Abstract Syntax of Policy

The top of Figure 11.2 shows an example specification in JPolCat for patient-related resources and permissions for roles in an example GP/doctor's surgery, with the resulting tables 'Resources', 'SCategories', 'DCategories' and 'CanBe' shown at the bottom.

```
Resource lifts = ["use"];
Resource emergencyExit = ["use"];
Resource patient = ["getName", "prescribe"];
Category admin = [(lifts, ["use"]), (patient, ["getName"])];
Category doctor subsumes [admin] = [(patient, ["prescribe"])];
Category* emergencyEmployee = [(emergencyExit, ["use"])];
admin can-be [emergencyEmployee];
doctor can-be [emergencyEmployee];
```

Resources	
name	actions
Lifts	use
Emergency Exit	use
Patient	getName
	prescribe

SCategories	
name	permissions
Admin	(Lifts, use)
	(Patient, getName)
Doctor	(Lifts, use)
	(Patient, getName)
	(Patient, prescribe)

DCategories	
name	permissions
Emergency Employee	(EmergencyExit, use)

CanBe	
can	be
Admin	EmergencyEmployee
Doctor	EmergencyEmployee

FIGURE 11.2: Example JPolCat Code and Tables Representation

11.2 Semantics

We can state the semantics of the policy language in a concise manner by mapping the abstract syntax to elements of the CBAC model (excluding user-to-category assignments) i.e. to resources, actions, categories, dynamic categories and permissions: there

is a one-to-one correspondence between the resources, actions, categories and permissions specified in JPolCat and the equivalent parts of the CBAC model. In particular, the *perms* statement in JPolCat syntax (see the grammar rule for *perms* above), used in statements *sCat, dCat, sCatSub, dCatSub*, corresponds directly to adding a permission to a declared category in the CBAC sense. Note that for the dynamic categories, the programmer must provide a reference monitor which implements a separate policy file that specifies the user-to-category assignments, which is beyond the scope of this thesis. We provide a pattern for the reference monitor to enforce the CBAC policy for dynamic categories.

Therefore, we can define policy satisfaction as follows.

Definition 11.1 (Policy Satisfaction)**.** A Java program satisfies a JPolCat policy if for any invocation $res.m$ that exists in the program, where res is an instance of a resource class Res and m an action, only authenticated users that are in category c, such that the JPolCat policy specifies the permission $[Res, m]$ for s, can perform it.

Chapter 12

Target Program Patterns

The ‘SessionModel’ class contains an instance of ‘SecurityContext’, which holds all the security relevant information as needed for the CBAC implementation i.e. the situational and event information occurring at run-time. The ‘SessionModel’ passes the user either to Static- or Dynamic-Category Interfaces. Both of these operate similarly to the RBAC MVC patterns (see Chapter 7) but with naming restrictions altered (i.e. classes start with the name of the category to which they belong instead of the role). In particular, Static Category Interfaces - classes beginning with ‘SCategory’ in Figure 12.1 - directly access resources, but Dynamic Category Interfaces - classes beginning with ‘DCategory’ in Figure 12.1 - must be checked at run-time by the ‘Categoriser’, which is the programmer-provided run-time in-lined reference-monitor class. However, the call to the ‘Categoriser’ class does not have to be done by the programmer, because at compile-time our verifier will generate the code to do the check in Dynamic Category Interfaces. All classes operate as discussed in Chapter 10.

Note that the restrictions needed to compensate for Java’s dynamic dispatch are still needed in these patterns (see Chapter 7.1).

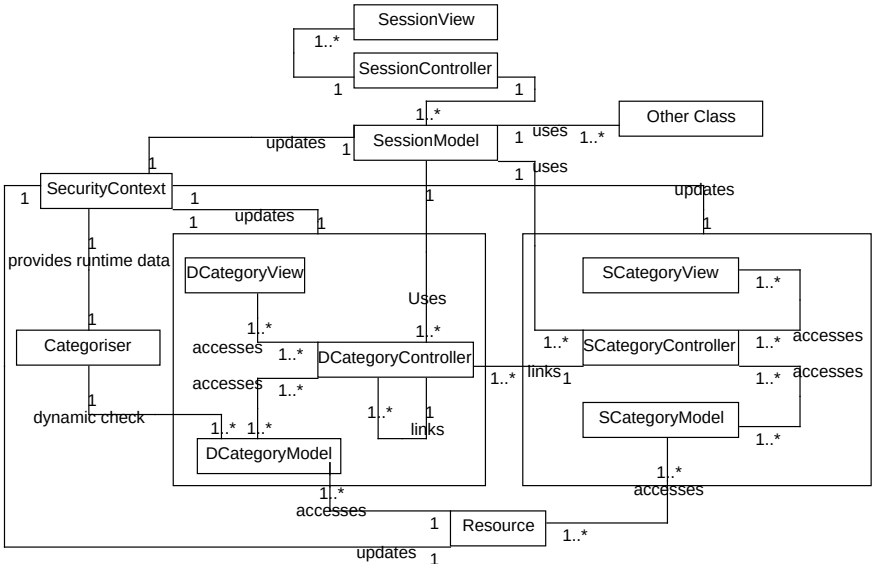

FIGURE 12.1: UML Class Diagram of CBAC MVC Patterns.

12.1 Security Context and Reference Monitor Patterns

We re-use previously proposed patterns for the security context and reference monitor, since they are well-studied in the literature (see, for example, [47] for the former and [20] for the latter). However, we do impose some restrictions on top of these patterns, in order to aid hybrid verification, as follows. Firstly, names of classes implementing the *Security Context* pattern must begin with the string "SecurityContext", with one class whose name is exactly "SecurityContext" - we will call the *main* security context class. This is needed so that the static verifier can identify and group the classes as 'SecurityContextClasses' - those which implement the security context. For the same reason, names of classes implementing the *Reference Monitor* pattern must begin with the string "Categoriser". This is name we use to represent a reference monitor for CBAC, which computes if the user is in a given category at run-time depending on the user-to-category assignment in the policy which use dynamic information. Additionally, there must be one class whose name is exactly "Categoriser", the *main* reference monitor class, containing a method with the following signature:

```
17 public boolean checkCategory(
18         SecurityContext securityContext,
19         String categoryToMatch)
```

LISTING 12.1: Code generated to invoke reference monitor

This method computes whether a user is a member of the (dynamic) category that has the same name as the parameter 'categoryToMatch'. This computation is to be done according to the user-to-category assignments in the policy utilising the dynamic information held in the parameter 'securityContext'. Note that any class could potentially need to update the security context, because the CBAC policy could use any number of a wide range of possible run-time information for the user-to-category assignments. Therefore, any class could need to hold a pointer to the main security context class as a data field, to ensure it can be accessed and updated in that class. The main reference monitor class will need to be accessible, as a data field, in all the classes which correspond to dynamic categories, so that a call to the reference monitor can be made before each action invocation.

12.2 Category Model-View-Controller (MVC) Pattern

- **Name**: Category Model-View-Controller
- **Example**: A system in which some classes represent resources much like the context as described in our RBAC MVC patterns e.g. a class each for tables in a GP Surgery database such as *Patients, Appointments, Staff*. However, access to invoking actions in each resource needs to be restricted using the CBAC model. This means that there are a number of categories in the policy that have a different set of permissions associated with them. Categories such as 'Admin' and 'Staff'

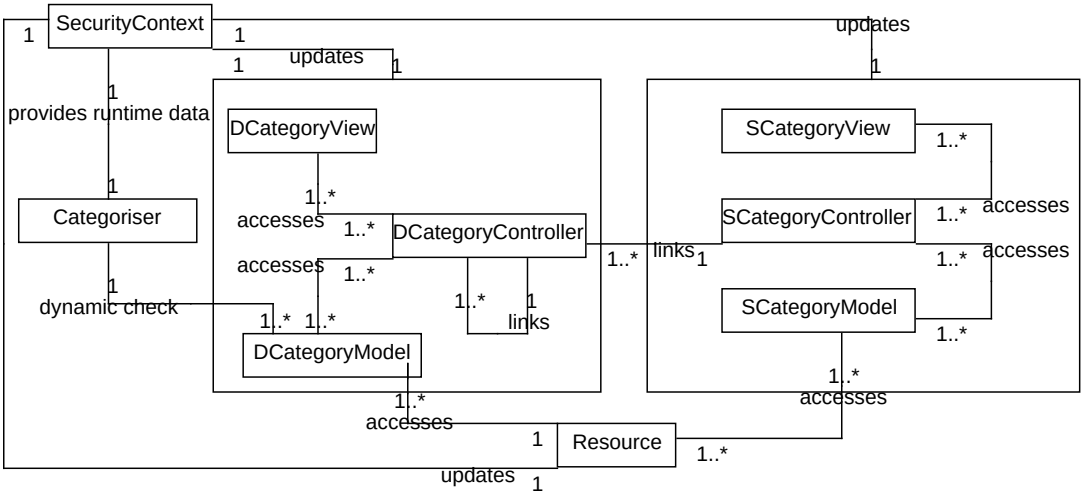

FIGURE 12.2: UML Class Diagram of Category MVC Pattern

would exist. Each category has a set of Tasks that it can do e.g. 'Admin' can do a Task 'Register Patient'.

- **Context**: An application in which access to resources needs to be restricted, following the CBAC model. In this case, the policy specifies restrictions on method invocations by categories, which can be either static or dynamic. A resource is a class (e.g. PatientsTable) which contains some methods that can only be invoked by roles that have been assigned the permission to do so (e.g.`read()`, `write()`, `delete()`). We call such methods *actions*. Any other methods in resources are called *auxiliaries*. So, both static and dynamic categories are associated with a set of permissions; the right to invoke a particular action of a resource. Assigning a category the permission to invoke an action on a resource means that the invocation can be on any of it's instances. The CBAC policy is to be enforced using a hybrid mechanism combining compile-time and run-time verification.

- **Problem**: The policy specifies permissions for static categories and dynamic categories to invoke actions on resources, however a user's interaction with the system will usually consist of performing a set of *tasks* in the system that are

dependent on their category/categories. The business logic part of the program needs to implement these tasks. In this part, when an action is invoked in the code, the static verifier must be able to know all the possible categories the user *could* be a member of at the point in the code where the invocation takes place. Then it can be checked if those categories have the permission to call that action according to the policy. At compile-time, the static verifier will decide if all invocations of actions in the program are valid according to the policy; the action invocations made in classes corresponding to categories are permitted for the corresponding category in the policy. This will be sufficient for static categories, since for these, the user-to-category assignments do not depend on run-time information. For dynamic categories, we first need a session-persistent data store for the run-time information. This is provided by the Security Context pattern (see Chapter 12.1). Then, at run-time, the reference monitor needs to intercept those action invocations where the user must be a member of a dynamic category. It must only allow them to be executed if the user is in a dynamic category with the permission to invoke that action. This is provided by the Reference Monitor pattern (see Chapter 12.1).

- **Structure**: The structure for the solution is shown in the UML Class diagram in Fig 12.2. It contains:
  - *Resource1*: a class representing a resource. One class for each kind of resource in the policy (referred to as *Resource1, Resource2, .., ResourceN*) is needed. The name must be exactly the name of the represented resource.
  - *StaticCategoryModel and DynamicCategoryModel*: the former is a Static Category Model component and the latter is a Dynamic Category Model component. These are similar to a "Model" class in the sense of the MVC pattern, but each one is associated to a single static category or dynamic

category, respectively. At least one model component for each category in the policy is needed (shown in the diagram as *SCategory1, SCategory2, .., SCategoryN* for static categories or *DCategory1, DCategory2, .., DCategoryN* dynamic categories). The name must be the name of the associated static or dynamic category, then followed by the string 'Model' and then optionally followed by any characters that will form a valid Java class identifier. The latter option allows for multiple model classes to be associated to a single category.

– *StaticCategoryController and DynamicCategoryController*: See the above *model* components, except these classes are similar to "Controller" components instead. The names are the name of the static or dynamic category, followed by the string 'Controller' (instead of 'Model') followed by any characters that will form a valid Java class identifier. *StaticCategoryView and DynamicCategoryView*: As above for category controller components, except these classes are similar to 'View' components. The names follow the same rules as above, except the string 'Controller' is replaced by 'View'.

- **Dynamics**: There are a number of factors in the dynamics of this pattern. We start by discussing the dynamics each of the category model, view and controller classes, then the relations between them and finally the relations with security context and reference monitor classes.

  – Firstly, the business logic part is contained in the Resource classes and Static/Dynamic Category Model classes (any classes not related to the policy can also included). Resource classes contain a set of action and auxiliary methods. Static/Dynamic Category Model classes contain a set of *task* methods which contain the business logic for carrying out tasks based on the category and can contain one or more action method invocations. In the business

logic part of the application, invocations of actions should only be made in Static/Dynamic Model classes and Resource classes.

- Next, the presentation of data and user-interaction, to allow the user to access and perform tasks, is to be implemented by the Static/Dynamic Category View classes. These may invoke actions, for example to display access-restricted data, and the category model classes which belong to the same category, to invoke its tasks.
- The direct access between view and model components is discouraged, however, to enable the flexibility that the MVC pattern facilitates. This is achieved in this pattern by the Static/Dynamic Category Controller classes, which act as a mediator between the models and views, processing events from both of these and selecting the model or view to use as a response. This allows models and views to be modified independently whilst preserving communication between them. The category controller may, if needed, invoke actions in resources and the category model classes belonging to the same category.
- Each category is associated to a set of category model, view and controller classes. These behave similarly to the standard MVC pattern. The relations between category MVC classes go further than the the standard MVC pattern, to reflect the CBAC specification. The flow of the program should initially allow the user to invoke the category MVC classes of a any of their policy-assigned static categories - this is to be implemented in the following pattern *CBAC Session*. From any chosen static category's MVC classes, they must be able to access the category MVC classes of any of the dynamic categories that are either hierarchically or *can-be* related to any user-chosen static category. From there, they must be able access the category MVC classes of any of the dynamic categories which are either hierarchically or

can-be related to the chosen dynamic category. Using this approach, the user can progressively go through the category MVC classes in the program to find the task that they wish to execute. To achieve this flow of relations between category MVC classes, we utilise the concept of a *link* (see Definition 10.3).

– Lastly, every resource and static category model class in the pattern *may* invoke classes that implement the SecurityContext pattern. This is so that the run-time information, held in these classes and used in the run-time policy enforcement, can be updated as needed. The Dynamic Category MVC classes *must* contain, as data fields, an instance of the main security context class and an instance of the main categoriser class. These fields are needed to ensure that a call to the categoriser can be made before each action invocation found in the class. Also, every link between category classes must ensure to pass the instance of the main security context class, so that it can be updated and used in the run-time checks, if needed.

- **Implementation**: In resource classes, action methods should have the modifier 'public' and auxiliary methods should have the 'private' modifier, so that they are not accessed by other classes. The reason for this is that we allow action invocations in resources, therefore if an auxiliary method invokes an action and the method is accessible to other classes, then other classes could indirectly call actions through the auxiliary. An example of an action and auxiliary within our example is as follows: if each of the classes had an action `findAll()`, it would have the modifier `public`. If each had an auxiliary method `format()`, it would have the `private` modifier.

  Static/Dynamic Category Model components should be implemented as classes with the same name as the role followed by the string 'Model', followed by an

optional string of characters which make up a valid Java class identifier e.g. for a Static Category 'Admin', one Static Category Model class name can be `AdminModel`. The Task methods should have the 'public' modifier. The controller and view components for this category can be `AdminController` and `AdminView`. In JEE, the model components can be implemented on the server-side, as Session EJBs. For any dynamic category MVC class that contains a field which is an instance of the main security context class this field must be named "securityContext" exactly. Also, the field which is an instance of the main categoriser class must be named "categoriser" exactly. This is so that at compile-time, the names of these are known without any computation required, in order for code to be generated which calls the categoriser's 'checkCategory(...)' method before each action invocation.

- **Consequences**: The resources and categories in the policy are implemented directly in the program. The parts of the functionality of the program, which are affected by the policy, are implemented in resource classes and category MVC classes. The latter implement this functionality through the task methods which are specific to each category. At compile-time, the resources and the category to which each MVC class belongs to can be easily known as a result of the naming restrictions on the class names. As a result, when there is an action invocation in the code of a category MVC class, we know that the category that this class belongs to *should* be the one that the user is a member of at run-time to be able to execute it. Note the use of *should*, because in order for us to be sure that the associated category is active at run-time, we need to ensure the following. For static categories, the flow of the program allows users to only reach the category MVC classes of those static categories that have been assigned to them in the policy. This will be addressed by the following pattern *Session CBAC* and as part of the static verification checks. For dynamic categories, when invoking an

action in a category MVC class belonging to a dynamic category, the user must be a member of that category at run-time (according to the user-to-category assignments in the policy). This will be addressed by the code generation phase by our compile-time code generator, which relies on the implementation of the Categoriser pattern.

Note that, as with our RBAC MVC patterns, actions can be invoked in Resource classes also, which is left unrestricted (in terms of both design and static analysis) in order to allow flexibility in implementing resources that depend on one another.

- **Known Uses**: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses CBAC. In this thesis, we show an implementation in JEE.
- **Related Patterns**: CBAC Session, Security Context, Reference Monitor.

12.3 CBAC Session

- **Name**: CBAC Session
- **Context**: The context is carried over from the previous pattern Category MVC (See Chapter 12.2), which is used in this pattern.
- **Problem**: The flow of the program must initialise a session (in it's traditional definition) with the user, by initialising the store of run-time session-scoped information, and then proceeding to authentication of the user. If successful, they must then be able to interact with the system via the category MVC classes of only those categories that have been assigned to them in the policy.

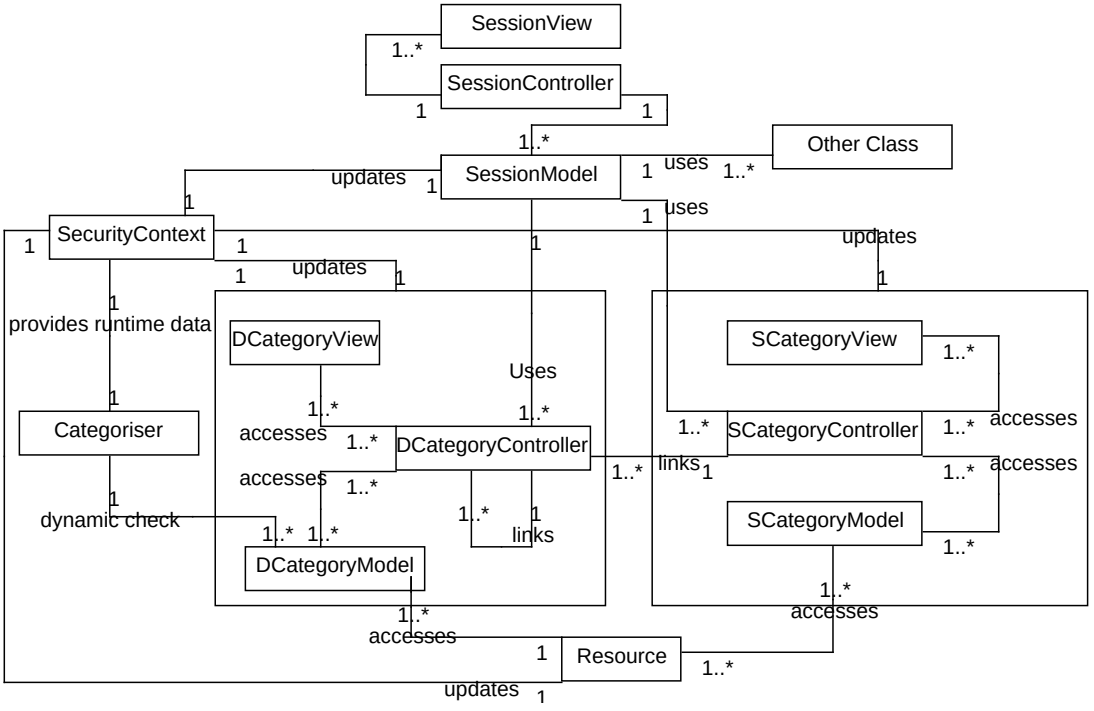

FIGURE 12.3: UML Class Diagram of CBAC Session Pattern

- **Structure**: The structure for the solution is shown in the UML Class diagram in Fig 12.3. We describe the elements that are in addition to the Category MVC pattern as follows:

  - *Session Model, Controller and View*: These are MVC classes that are specific to the session. The session model implements the session tasks as its methods. The session views handle user-interaction for the session tasks. The session controller acts as a mediator between the views and models, allowing both to be modified independently. The names of these classes must start with 'Session', followed by either 'Model', 'View' or 'Controller', then optionally followed by any string that will form a valid Java class identifier. As such, there can be more than one of each of these classes.

  - *Other Class*: These are classes that the policy is not concerned with. They implement *other tasks*.

- **Dynamics**: The application begins by initialising the session MVC classes and displaying its interface. The session interface first initialises the SecurityContext classes and holds an instance of the main SecurityContext class. The Session also handles authentication, which is not shown in this pattern because it is not dealt with in our approach. After successful authentication, a list of static categories, that have been assigned to the user in the policy, is to be retrieved. The design and implementation of the authentication process is left open to the programmer, so that our approach allows flexibility in regard. The session invokes the static category classes, specifically the category controller, of a user-selected static category from the retrieved list. This invocation results in that category's interface being displayed to the user. It also passes the instance of the main SecurityContext class to the category MVC classes so they can update the run-time information, if needed. The session interface also invokes 'other' classes, which are not related to the policy. It can direct the user to the other classes if they choose to perform other tasks.

- **Implementation**: The session interface must be the first interface displayed to the user. Also, it must always be active. One example of achieving this is to have the session interface as a parent frame, with the remaining interfaces in the program (other classes, category interfaces) implemented as internal windows. The session tasks can be implemented as menu items in the parent frame.

- **Consequences**: The Session MVC classes provide a complete interface specifically for the implementation of the session tasks. This interface should always be active and visible to the user so that they can access the session tasks at any time. The interface allows users to perform all the necessary session tasks e.g. authentication, logout, activating a static category interface and performing an other task. The user will be authenticated and if successful, will be able to select

one of their retrieved static category interfaces to activate. From there, they can reach the dynamic category interfaces of the dynamic categories related to the activated static category, which is a result of the previous pattern, Category MVC. If the session tasks are implemented correctly, then we can assume that the user will only be able to activate the static categories that have been assigned to them in the policy. This aids the process of determining statically that in order for a user to reach an action invocation in a static category interface, they must be a member of that category according to the policy. However, checks are needed at compile-time to ensure this, because there are still ways to reach the an invocation in a static category interface without activating the associated category, for example if an other class invokes a static category interface.

- **Known Uses**: This pattern can be used in any OO language that uses visibility modifiers, specifically 'public' and 'private', where the access control policy uses CBAC. In this thesis, we show an implementation in JEE.

- **Related Patterns**: CBAC Session, Security Context, Reference Monitor.

Chapter 13

Hybrid Enforcement Mechanism

Our source level static analysis (static verification and code generation) takes as input a well-formed program, which is defined as follows:

Definition 13.1 (Well-formed program)**.** A well-formed program consists of a (syntactically correct) JPolCat policy file and a Java program that implements the CBAC MVC patterns defined in Chapter 12. Implementing the patterns means the following. There is a set of session classes (one model, one controller and multiple views associated to the session), a set of resource classes, a set of static category interface classes (sets of one static category model, one static category controller and multiple static category view classes), a set of dynamic category interface classes (sets of one dynamic category model, one dynamic category controller and multiple dynamic category view classes), a set of security context classes, a set of categoriser classes, and a set of classes which do not fit into the other groups. These correspond to the classes shown in the design pattern. In particular for session classes, they: correctly authenticate users, retrieve the correct static categories that have been assigned to the user in the policy, activate the correct static categories allowed for the user, switch static categories correctly for

the retrieved and selected static category. Lastly, inheritance is disabled between: resources, two category MVC classes belonging to different categories and a resource, a category MVC class, a session class, a security context class and an 'other' class.

The static verifier should reject a program if an access violation is found, else accept it. In other words, it should only accept a program that will be compiled into a program enforcing the policy (see Definition 11.1). In this chapter, we describe high-level details of the algorithm for static checking and code generation. The first of three phases, described in Chapter 13.1, generates abstract representations of the policy and program, and populates tables to be used in the second phase. The second phase, described in Chapter 13.2, uses the abstract representations and tables to check that the program respects the (static parts of) the policy. The third phase, described in Chapter 13.3 generates code in dynamic category interface classes to call the categoriser before each access request that occurs. Each phase is described as follows.

13.1 Grouping Classes

The program is parsed in order to identify and group the different kinds of classes in the program. Each class is stored in a separate table depending on the group, but all classes have the same representation; its name, a collection of the methods declared in it and a collection of all method invocations made in it. This is the data needed for us to check method declarations and method invocations i.e. call graph analysis. To group the classes, the naming conventions briefly discussed in Chapter 12 are used. The grouping process is exactly the same as in the previous case for static RBAC enforcement, except, of course, the modifications required due to the concept of role having been replaced with static and dynamic category. Hence, instead of matching a role MVC class name prefix with a role in the policy, this case matches category MVC

class name prefixes with either static or a dynamic category in the policy. This results in a group of static category MVC classes and a group of dynamic mvc category classes (instead of role MVC classes). Also, this case adds classes for the SecurityContext and ReferenceMonitor pattern implementation. The grouping process again works very similarly to other groups - any class whose name starts with 'SecurityContext' is a security context class and any class whose name starts with 'Categoriser' is a categoriser class.

As an example, we can discuss at a high level how Dynamic Category Interface Classes are identified and grouped. Iterating through all the classes in the program, if the name of a class starts with the name of a dynamic category - checked by looking at the data generated from parsing the policy - and the subsequent characters to the end of the name are 'Model' then it is a Dynamic Category Model class.

At the end of the grouping phase, the program parser generates a table for each of the groups of classes: 'OtherClasses' 'SessionClasses', 'SecurityContextClasses', 'CategoriserClasses', 'ResourceClasses', 'DCategoryModelClasses', 'DCategoryControllerClasses', 'DCategoryViewClasses', 'SCategoryModelClasses', 'SCategoryControllerClasses' and 'SCategoryViewClasses'.

During static verification, the invocations made in each type of class are subject to a different set of checks. We frequently refer to the checks described in the previous, static, system 8.3 to avoid repetition.

13.2 Static Verifier Checks

We briefly discuss the checks made in the different groups of classes.

13.2.1 Resource Classes

The checks made in classes in 'ResourceClasses' are the same as the previous work, but now there are more tables of classes that Resource classes cannot invoke methods in. To simplify, invocations of methods in classes belonging to the tables *SecurityContext* and *Categoriser* are allowed only, as well as to other Resource classes. For the former two, this is so that resource classes can update the security context information, if needed, and initialise the categoriser. For the latter, this is to reduce the restrictions for programmers when implementing resources which may depend on each other.

13.2.2 Static Category MVC Classes

The checks made in Static Category MVC classes (i.e. classes in the latter three tables above) are the same as the checks in the previous work (for Role MVC classes) for invocations of classes in the table 'ResourceClasses' and other (dynamic and static) category MVC classes. However, for the latter, we allow calls to controller or model class of any dynamic category that is *can-be* related to the owning category of this clas, according to the 'CanBe' table generated from the policy. For invocations not on Category MVC classes, this version of the verifier ensures no invocations are made to methods in classes in any of the other tables.

13.2.3 Dynamic Category MVC Classes

The checks for the three tables beginning with 'DCategory' above are equivalent to the checks above in Chapter 13.2.2, except that Dynamic Category MVC classes can call on classes in the table 'CategoriserClasses' for the method *checkCategory()*.

13.2.4 Remaining Groups

In the remaining tables, no classes can invoke methods of classes in any other group, except in the following cases. Classes in the table 'SessionClasses' can invoke methods in 'SCategoryControllerClasses', 'DCategoryControllerClasses' and 'OtherClasses' - to pass the flow of the program to a category interface or an other class. They can also invoke classes in the table 'SecurityContextClasses' to update the run-time information stored in those classes.

13.3 Generating Code

The classes in tables beginning with 'DCategory' - dynamic category interface classes - are then checked to find invocations of actions on resources. When in an invocation $res.m$ is found (where res is an instance of a Resource class and m is an action), the entire body of the method in which the action invocation occurs is replaced with the following code:

```
17 if(categoriser.checkCategory(securityContext,
18      theCategory) == true))
19           {
20           ...entire method body...
21           }
22 else throw new
23      AccessControlException(``Access denied.");
```

Listing 13.1: Code generated to invoke reference monitor

Here, 'categoriser' is a variable of type 'Categoriser' given in the design pattern, which must contain a method 'checkCategory()' (as described in the pattern in the long version of our paper). This takes the three parameters: a variable 'securityContext'

of type 'SecurityContext' given in the pattern, 'theCategory' which is a string that is the name of the category which is the owner of the class in which the action is invoked and '...entire method body...' is the code in the body of the method before any modifications by our code generator. The variables 'categoriser' and 'securityContext' must be made visible in dynamic category MVC classes. The 'checkCategory' method is implemented by the programmer, in the 'Categoriser' class which they provide, with no restrictions imposed in our approach to comply with the generality required by CBAC. Our assumption is that this method always returns the correct result when checking if a user has the required category to execute an action depending on the situational and event information in the Security Context at run-time. The result will be 'true' to grant the request, or 'false' to deny it.

To summarise, all category interface classes will only be allowed to call those actions that are allowed according to the permission-to-category assignments in the policy. This is sufficient for static category interface classes, but dynamic category interface classes will additionally have code generated to check at run-time if the user has the required category to invoke an action according to the user-to-category assignments in the policy.

13.4 Properties

The static verification combined with the code generation ensures action invocations are checked at either compile-time or run-time to ensure they only execute if they are valid according to the policy. More precisely, programs satisfy the following properties.

Definition 13.2 (OK program)**.** A program which implements the design patterns and fulfils the following conditions. Category MVC classes: invoke only those actions allowed for the associated category according to the JPolCat policy file, invoke only

those category controller classes that the owning category is *can-be* related to and no other category MVC classes, do not invoke classes from any other group except that every action invocation in a dynamic category interface class is inside the true branch of a conditional statement which is only executed if a call to the request checking method of the categoriser class returns true, signifying that the action is allowed according to the CBAC policy. Resource classes do not invoke any classes from any other group of classes. Session classes: do not invoke classes from any other group except category controller classes and security context classes and other classes. The remaining groups do not invoke methods in classes belonging to any other group.

Proposition 13.3. *Let P be a well-formed program. P is accepted by the verifier if and only if $OK(P)$.*

Proposition 13.4. *A well-formed program P accepted by the verifier satisfies the policy (see Definition 11.1).*

We provide an intuitive explanation of the propositions as follows. According to Definition 11.1 we need to show that only authorised users belonging to category c having permission $[Res, m]$ can invoke the action m of an instance of Res. Let $res.m$ be a call to m in the program P, for which the parser has identified the called class to be Res and the called method to be an action m. Since P is *well-formed*, by Definition 13.1 it implements the CBAC MVC patterns. Then, a user u can only execute $res.m$ if the user has been authenticated and is in a session, where either one of u's static categories, say s, has been activated or, the flow of the program has passed to a dynamic category interface.

If a static category has been activated, then this implies that the user is now interacting with s's static category interface. From this, the user may then select to view the dynamic category interface of a dynamic category x where $c \rightarrow x$ due to Definition 10.3.

The same applies if the user is interacting with the interface of a dynamic category d. Moreover, P has been accepted by the verifier, by Theorem 13.3, $OK(P)$. From these two facts, we deduce, by Definition 13.2, two conclusions. Firstly, if in a static category interface for category s, only the invocations $res.m$ authorised for the category s can be performed (there are no unauthorised calls to actions) since no invalid invocations exist in the source code. Secondly, if in a dynamic category interface of category d, when the user invokes an an action $res.m$, firstly $res.m$ will have been checked at compile-time that d is authorised to invoke it and secondly, code will have been generated to check at run-time, before the invocation, if the user is a member of any category with the permission to invoke $res.m$.

Chapter 14

Case Study

We extend our previous system's case study, adding two dynamic categories *NHSNurse* and *PrivateNurse*.

14.1 Policy

We respecify the JPol RBAC policy from the previous system, this time in JPolCat with the roles as static categories. Then we add the dynamic categories to the specification. An extract of the JPolCat policy for resources is below, followed by an extract for the categories. There are two things to note from inspecting the policy. Firstly, the syntax is far more compact than JPol. Secondly, as shown in Listing 14.3, when specifying dynamic categories, the programmer must ensure that every dynamic category is in a hierarchy where it can be reached from a static category, or that it is reachable from a static category in the set of *can-be* relations in the policy. This is a necessity for the implementation of the flow of the program; interfaces of static categories can be directly invoked from the session interface however, dynamic category interfaces can only be reached through static category interfaces or another dynamic category interface. So,

this restriction is needed to ensure that the dynamic category interfaces are reachable in the program flow. In this case study, this restriction is satisfied by creating a new, empty static category *Nurse*. This is then *can-be* related to the two dynamic categories *NHSNurse* and *PrivateNurse*.

Lastly, we briefly explain the user-to-category assignments for the dynamic categories, which aid the development of the program. This is because we can ensure to implement and store the dynamic information that is needed in the security context. The information always contains at least the username of the logged in user, but additionally in this case, the current day. Users are assigned to the two dynamic categories depending on the day (it has its usual seven values). A user *bob* is assigned to 'NHSNurse' if the day is 'Monday', 'Tuesday' or 'Wednesday'. Also, *bob* is assigned to 'PrivateNurse' if the day is 'Thursday' or 'Friday'.

```
1  // (1) Declare Resource ``Nhspatient" and declare its actions
2  Resource Nhspatient =
3          [Nhspatient, setFirstname, getFirstname,
4          setLastname, getLastname, setDob, getDob,
5          setNhsnumber, getNhsnumber,
6          setPatientid, getPatientid
7  // ...add more actions...
8          ];
9  // Do (1) for each Resource
10
11 Resource Privatepatient =
12         [Privatepatient, setFirstname, getFirstname,
13         setLastname, getLastname, setDob, getDob,
14         setPaymentdetails, getPaymentdetails,
15         setPatientid, getPatientid
16         ];
17
18
```

```
19 Resource NhspatientsFacade =
20         [NhspatientsFacade, create, edit, remove,
21         findAll, find, count, generatePid
22         ];
23
24 Resource PrivatepatientsFacade =
25         [PrivatepatientsFacade, create, edit,
26         remove, findAll, find, count, generatePid
27         ];
```

LISTING 14.1: Extract of Sample JPolCat Policy declaring Resources with their actions

```
1
2  // (2) Declare Static Category ``NHSDoctor" and
3  // add each of its permissions
4  Category NHSDoctor =
5          [(Nhspatient, getFirstname), (Nhspatient, setFirstname),
6          (Nhspatient, getLastname), (Nhspatient, setLastname),
7          (Nhspatient, getDob), (Nhspatient, setDob),
8          (Nhspatient, getNhsnumber), (Nhspatient, setNhsnumber),
9          (Nhspatient, getPatientid), (Nhspatient, setPatientid),
10         (NhspatientsFacade, findAll), (NhspatientsFacade, find),
11         (NhspatientsFacade, create), (NhspatientsFacade, remove)
12         //... add more permissions ...
13         ];
14 // Do (2) for each Category
15
16 Category PrivateDoctor =
17         [(Privatepatient, getFirstname),
18          (Privatepatient, setFirstname),
19          (Privatepatient, getLastname),
20          (Privatepatient, setLastname),
21          (Privatepatient, getDob),
```

```
22          (Privatepatient, setDob),
23          (Privatepatient, getPaymentdetails),
24          (Privatepatient, setPaymentdetails),
25          (Privatepatient, getPatientid),
26          (Privatepatient, setPatientid),
27          (PrivatepatientsFacade, findAll),
28          (PrivatepatientsFacade, find)
29         ];
```

LISTING 14.2: Extract of sample JPolCat policy declaring static categories with their permissions

```
1  //Declare the Dynamic Categories ``NHSNurse" and ``PrivateNurse"
2  Category* NHSNurse =
3          [(Nhspatient, getFirstname), (Nhspatient, setFirstname),
4           (Nhspatient, getLastname), (Nhspatient, setLastname),
5           (Nhspatient, getDob), (Nhspatient, setDob),
6           (Nhspatient, getNhsnumber), (Nhspatient, setNhsnumber),
7           (Nhspatient, getPatientid), (Nhspatient, setPatientid),
8           (NhspatientsFacade, findAll), (NhspatientsFacade, find)
9          ];
10
11
12 Category* PrivateNurse =
13         [(Privatepatient, getFirstname),
14          (Privatepatient, setFirstname),
15          (Privatepatient, getLastname),
16          (Privatepatient, setLastname),
17          (Privatepatient, getDob),
18          (Privatepatient, setDob),
19          (Privatepatient, getPaymentdetails),
20          (Privatepatient, setPaymentdetails),
21          (Privatepatient, getPatientid),
22          (Privatepatient, setPatientid),
```

```
23          (PrivatepatientsFacade, findAll),
24          (PrivatepatientsFacade, find)
25         ];
26 // Declare the empty Static Category ``Nurse"
27 // which is used to invoke the dynamic categories
28 // ``NHSNurse" and ``PrivateNurse" using the can-be relation
29 Category Nurse;
30
31 // Make ``Nurse" can-be related to the two nurses
32 Nurse can-be NHSNurse;
33 Nurse can-be PrivateNurse;
```

LISTING 14.3: Extract of sample JPolCat policy declaring dynamic categories with their permissions

14.2 Target Program

The previous role MVC classes are copied, since they are already static category MVC classes. Next, we add dynamic category MVC classes for these two dynamic categories. As before, all category model classes are implemented as session EJBS and stored on the server. Their category controller and MVC classes are implemented on the client-side, in an application client. Following the same structure, we then created a minimal static category interface for the category 'Nurse', since all it needs to do is invoke the interfaces of dynamic categories 'NHSNurse' and 'PrivateNurse'. Thus far, the implementation described corresponds to our Category MVC pattern. The session interface is created next, following our CBAC Session pattern, but its three MVC classes are stored entirely on the client-side in the application client. Then, we create the classes 'SecurityContext' and 'Categoriser', both using generic security context and in-lined reference monitor patterns, as session EJBs.

The former - 'SecurityContext' - contains a Java HashMap, storing dynamic information as a mapped pair *(key → value)* where every key points to a value. The keys are names of the information, stored as string types, and values are the values of type Object, since the value can be of a range of types. We then ensure every class that needs to update/use/pass on the dynamic information contains an instance of the 'SecurityContext' class, as a field called 'securityContext', and passes on this instance in the methods that need access to it. This is as usually specified in patterns for the security context, with the additional restrictions described in our pattern. At least all the dynamic category interface classes need to contain the instance of the security context. In our case, only the session MVC classes need to update the session, since the username of the logged in user and the system date will need to be written to the security context. The classes that will use it are also the 'Nurse', 'NHSNurse' and 'PrivateNurse' category MVC classes.

A similar process is undertaken for the 'Categoriser'. This class contains the method 'public boolean checkCategory(SecurityContext context, String categoryToMatch)'. This method checks the security context, extracting the date and the username of the logged in user. It then returns true in the following two cases. Firstly, the day (computed from the date) is 'Monday' or 'Tuesday' or 'Wednesday' and the username is 'bob' and the the 'categoryToMatch' is 'NHSNurse'. Secondly, the day is 'Thursday' or 'Friday' and the username is 'bob' and the the 'categoryToMatch' is 'PrivateNurse'. All dynamic category classes need to contain an instance of the 'Categoriser' class, as a field called 'categoriser'. This field needs to be correctly assigned when the objects are created. We do not need to do any more to any of the classes because code to call the categoriser will be generated in the program by our tool.

14.3 Static Checks: Extended

Most of the static checks described in Chapter 13 are very similar to those in the previous version of the system. In these cases, they are adapted versions of the previous checks, for example now there are more groups of classes that resource classes cannot call. For those kind of checks, refer to the previous system's case study. There is only one check that can be described as original in this system and this is discussed below.

14.3.1 Invalid Invocation Between Categories

In this system, invocations between category MVC classes are allowed, but only if the category of the caller is either hierarchically or *can-be* related to the callee. In Figure 14.1, we show an example of this case, where a category MVC class belonging to the category 'NHSNurse' invokes the a category MVC class of the 'PrivateNurse'. As shown in the JPolCat policy above, the first category does not have either of the relations with the second.

14.4 Code Generation

If the program passes all the static checks, then the verifier moves on to its next stage which is generating calls to the categoriser before action invocations in dynamic category MVC classes. Every method invocation in the dynamic category MVC classes is checked again, this time only to compute if it is an action invocation i.e. is the called class a resource and is the called method an action of that resource. If, in a method, an action invocation is found, we traverse up the AST of the program until we reach the root AST node of the entire method body. Then, we copy the entire method body AST and put it into the true branch of a newly created AST of the *if* statement shown

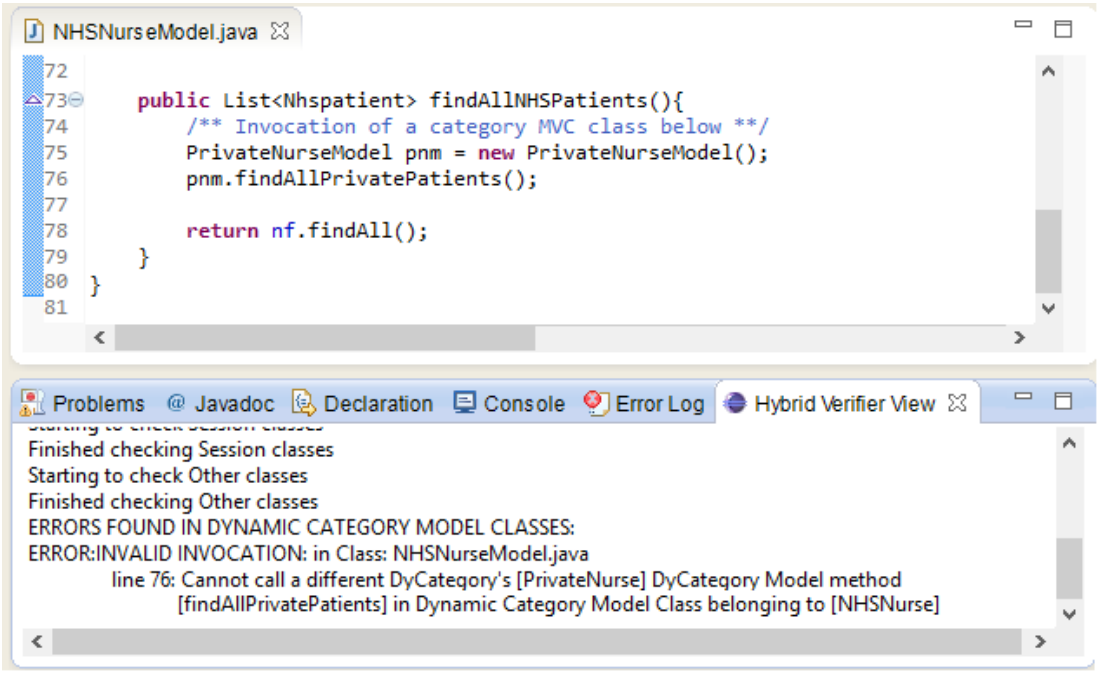

FIGURE 14.1: Example of error message shown when an invalid invocation between categories is found.

in Figure 13.3. We then replace the old method body AST with our new *if* statement AST and update the code of the class. We do this only once for each method, in which an action invocation is found. This prevents the class from having compilation errors, in cases where the action invocation is part of an extended line of code e.g. an assignment or *if* statement condition. Figure 14.2 shows the code in a class before code is generated on the top, then afterwards on the bottom.

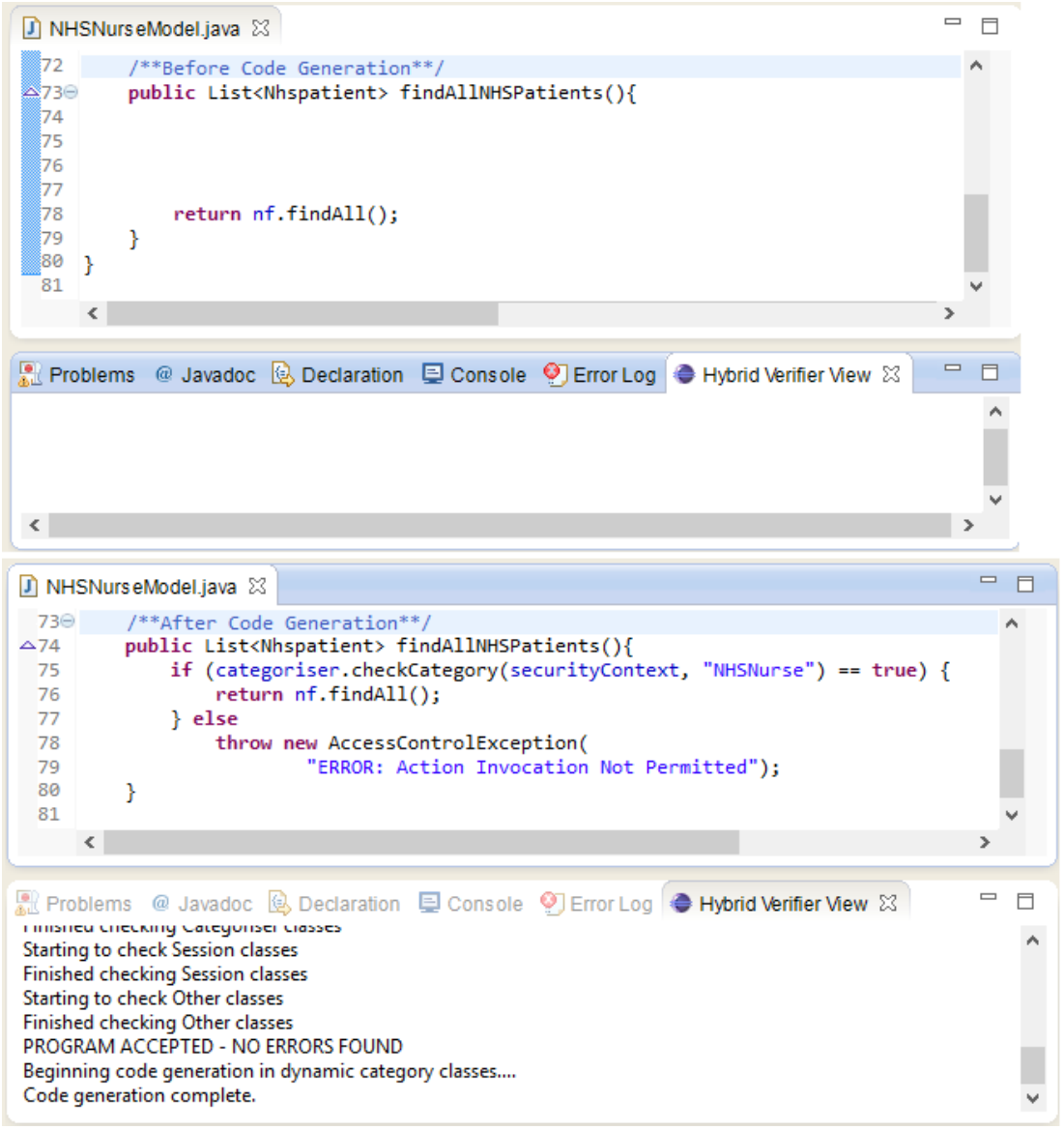

FIGURE 14.2: A class before and after code generation. The comment is shown for illustrative purposes only.

Part IV

Implementation and Evaluation

Chapter 15

Implementation Techniques

We chose to develop our verifier as an Eclipse [19] Plug-in, primarily because these are able to use the Java Development Tools (JDT) Application Programming Interface (API) provided by Eclipse. This API simplifies static code analysis because it represents Java programs as an AST, that can be programmatically accessed by the programmer. In Java, there are three ways to invoke a method; either invoking a ('static') method on a class e.g. 'ClassName.methodOne()', invoking a method on an object e.g. 'x.methodOne()' or invoking a method on the object returned by another method call e.g. 'x.methodOne().methodTwo()'. Accessing method invocations is a simple matter of finding the nodes in the AST of a class that correspond to method invocations. However, the challenge is to compute which type (i.e. class) an object, on which an method invocation is made, is an instance of. The JDT API can compute the type binding for variables and method invocation expressions, but due to dynamic dispatch, this can only be a very close approximation. To compensate, inheritance is restricted in our approach, therefore the computation of the type binding will be accurate. So, we can check if the object, on which an invocation is made, is an instance of a resource class, and therefore if the method being invoked is an action. This is sufficient

for all the checks which look for action invocations e.g. checking if a role class has the permission to invoke an action that is invoked in the class. In the same way, we can check if the object is an instance of a role class, to check if one role's components invokes another role's components. This is sufficient to implement all the static checks discussed in Chapter 8.3 and the static checks in the hybrid case.

Additionally, the JDT API allows the AST of a class to be edited by the programmer, by replacing a specific node in the AST with another node or subtree of nodes. Using this technique, we create the subtree that corresponds to the code that needs to be generated for the dynamic run-time check, discussed in Chapter 13. Then, when we find an action invocation, we add the node corresponding to the entire method body of the method in which it is found, into the place in the node in the subtree that corresponds to the true branch. We then replace the entire method body of that method in the AST of the class with our subtree. This fully generates the code into the program for our dynamic mechanism.

Chapter 16

Evaluation

The approach proposed in this thesis is for the development of web-based applications with RBAC or CBAC policies, based on the use of RBAC MVC or CBAC MVC patterns and a static enforcement algorithm and code generation for dynamic checks. It provides an easy-to-use methodology for programmers to integrate RBAC or CBAC policies into their applications and enforce them statically or using a hybrid approach. It can be applied to a variety of languages, not just JEE, by small modifications to the static verifier that are dependent on the details of the chosen language. For example, the naming restrictions in Chapter 13 would have to be altered depending on the naming conventions of the chosen language. Also, the concrete syntax of the JPol and JPolCat can be modified depending on the needs of the user/policy author/programmer. The version presented in the first system, JPol, is similar to Java, but it could be altered to follow any OO language syntax, or even a syntax closer to natural language which would help a policy author that is uncomfortable or unfamiliar with programming syntax. This is demonstrated by the modified syntax shown used in JPolCat, which is much more compact and concise. Additionally, the RBAC MVC and CBAC MVC patterns are already independent of implementation language. The only requirement

is that the chosen language must have access modifiers for methods, examples of which include C++ and C#.

The use of patterns to guide the implementation towards a program that can be statically verified for policy violations is a strength of our approach. Generally, patterns provide a solution to a particular recurring design problem. As such, adding access control into patterns for the design of the target program yields a uniform methodology to design the application integrating access control into the business logic. Access control is usually added on-top of an application at the last stage, enforced using a reference monitor, whereas using these patterns can guide the integration of the access control into the code. As a result, the program can be verified using a combination of static and dynamic mechanisms.

The drawbacks of using patterns include that the approach is less flexible since the program must follow the rules of the pattern. This can, however, be said of any approach that relies on the use of specific patterns. One particular drawback we encountered was that coding Tasks in Role/Category Model classes often resulted in duplicated code when two or more Roles could do the same task. A simple workaround mitigated this; a new Resource was added, and the Task was copied into the new Resource, effectively converting it into an action. The policy was updated to then specify which roles could call this action, and the Role Model classes were updated so that the old Task contained only a call to the new action.

The static enforcement that is used significantly in both of our systems mitigates several of the disadvantages of using a reference monitor. One of these is that there are less run-time overheads which are attributed to completely dynamic enforcement. The static enforcement checks do not impact on run-time resources whatsoever, since all access checks are done at compile-time. Another is that some mistakes in the access control policy can be caught earlier, at compile-time, which is preferable to discovering

mistakes at run-time. The static approach is, however, less flexible than the dynamic approach. Enforcing a policy at compile-time means that a policy cannot be modified at run-time. This means that that administrative changes to the policy, including changing, adding or removing resources and their action and roles/categories and their permissions, requires recompiling and verifying of the program. This is a limitation of the static approach which we use for part of our approach, rather than a direct limitation of our approach.

For our first system, which targets static RBAC policies, this restriction fine because these aspects of the policy do not change as much as the user-role or user-category assignments. Our approach does not restricts these, so whilst resource, role/category and permission specifications cannot be modified, users can still be assigned different roles/categories at run-time as job assignments change within the organisation.

We mitigate the limitations of the static approach by using a hybrid approach in our second system. The main strength of this work is that it can leverage the benefits of both static and dynamic enforcement approaches. The programmer has assurances at compile-time that the policy is enforced correctly in the program, significantly reducing testing time and making programs easier to debug, but the policy is still able to express dynamic information which is enforced, dynamically, also. The enforcement impact of the dynamic mechanism is reduced when compared with a traditional reference monitor. This is because it is in-lined into the program rather than being a separate program and also only those dynamic checks needed after the static checks have been made will be done, which will be fewer. Another strength is that because the policy in our approach follows the CBAC meta-model, it can be used in any environment where a specialisation of CBAC is used to specify the policy, of which there are a multitude [10].

There is another additional limitation of the static approach that occurs when analysing particular OO languages, particularly Java. It is a result of Java's Dynamic Dispatch. In any process that analyses method invocations, it is difficult to be sure of the exact class that the a method is invoked on i.e. for an invocation $x.method$, the class that x is an object of is difficult to compute, precisely, at compile-time (described in Chapter 4.1.4. Our current implementation computes this to be the declared type of variable x. However, at run-time, this will depend on the class of the object stored in x, rather than its declared type, which is computed only at compile-time using dynamic dispatch. We handle this at this stage by disabling inheritance between resource classes, and also between groups of classes. The former prevents actions from indirectly being invoked from another resource through inherited methods. The latter ensures the grouping is enforced so each class is subjected to the correct group's checks.

A weakness of our approach, in the second system, is that the programmer/policy author has to specify the user-to-category assignments separately and provide a reference monitor which implements this to our system. However, we identified this as a necessity due to the generality required by CBAC; the dynamic information used in the CBAC policy is completely variable and so it is difficult to generate a complete reference monitor at compile time using this.

Furthermore, the JPol/JPolCat policy is currently only verified to check that it is syntactically correct. In future work, we will apply formal verification to policies in JPol/JPolCat to ensure important properties such as completeness and consistency.

Lastly, note that user authentication is out-of-scope in our approach; we leave the implementation of user authentication to the programmer to allow for flexibility in this regard and in the assignments of users to roles. The user authentication can be supplied as module to our system in the part of the program which implements the

session pattern. We make the assumption that this module correctly authenticates users (as intended by the programmer).

16.1 Performance

An important property of static verification algorithms is their time complexity. For our verifier, where n is the number of classes in the program given as input to the verifier, the time complexity is of polynomial time, $O(n^5)$. This is the complexity for both the first static system and the hybrid system. This is due to the largest number of nested loops, which occurs when checking if an action invocation found in a category class is permitted by the policy. This nesting depth is the same for both systems.

We have run a number of tests to analyse the run-time performance of the verifier (implemented as an Eclipse plug-in). The programs tested were dummy programs which were generated for the purpose of the evaluation. Creating fully-functioning, realistic programs or utilising programs already created is not feasible at this stage, due to the fact that our proposed design patterns mean that programs must be developed with a new design methodology. We intend to investigate re-creating an existing program using our design patterns in future work.

Our tests identified the time taken, in milliseconds, to perform the static verification phase, the code generation phase and the overall time. We started with a base program consisting initially of: 10 session classes, 10 security context classes, 10 categoriser classes, 10 other classes, 10 resources, 10 static categories, 10 dynamic categories (130 total classes), 1 method per class (130 total methods) and 2 calls per method (260 total calls). In each of our sets of tests (discussed below), we increased one of these values while keeping the others the same. A policy file was also generated to match each test, such that each category is permitted to call every action in every resource.

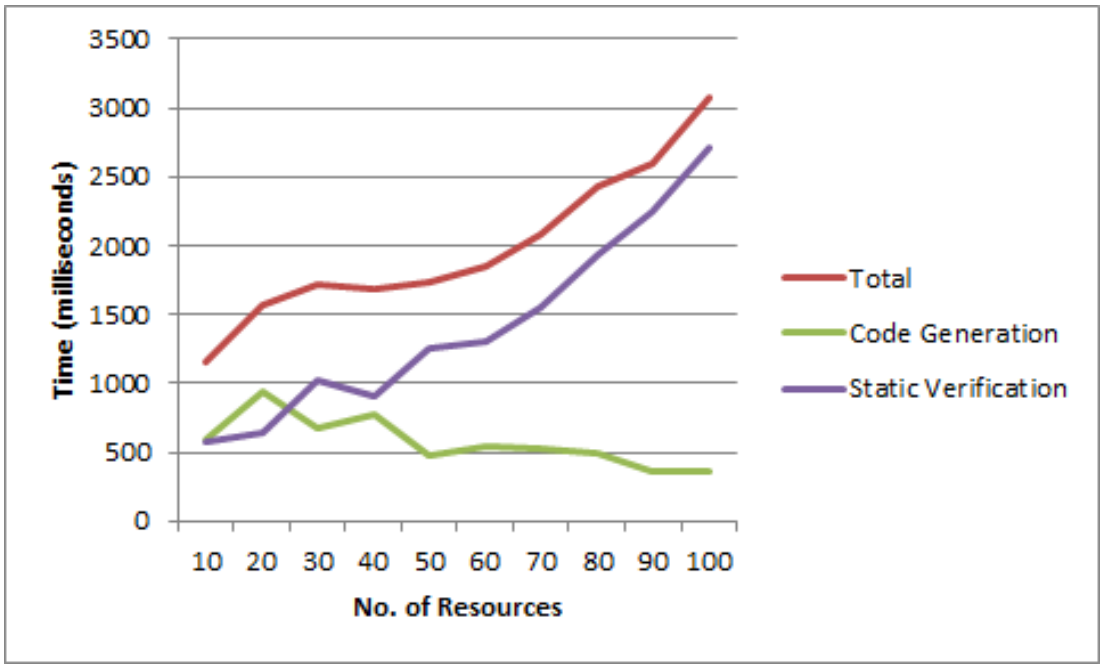

FIGURE 16.1: Runtime Performance Graph: Number of Resources

The programs contained some calls to actions, but none of which violated the policy or any of the other static verifier checks. The impact on running time when increasing the number of resources is shown in the graph in Figure 16.1.

The effect on performance by increasing the number of static categories is shown in the graph in Figure 16.2.

The impact on running time when increasing the number of dynamic categories is shown in the graph in Figure 16.3.

The effect on performance by increasing the number of invocations/calls in the program is shown in the graph in Figure 16.4.

The impact on running time when increasing the number of methods is shown in the graph in Figure 16.5.

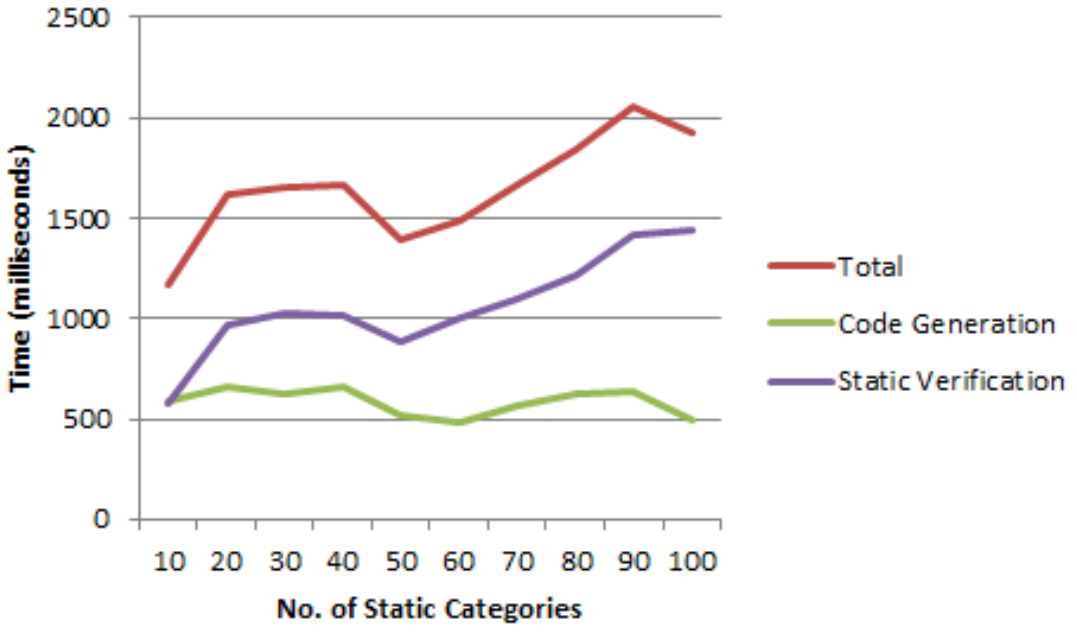

FIGURE 16.2: Runtime Performance Graph: Number of Static Categories

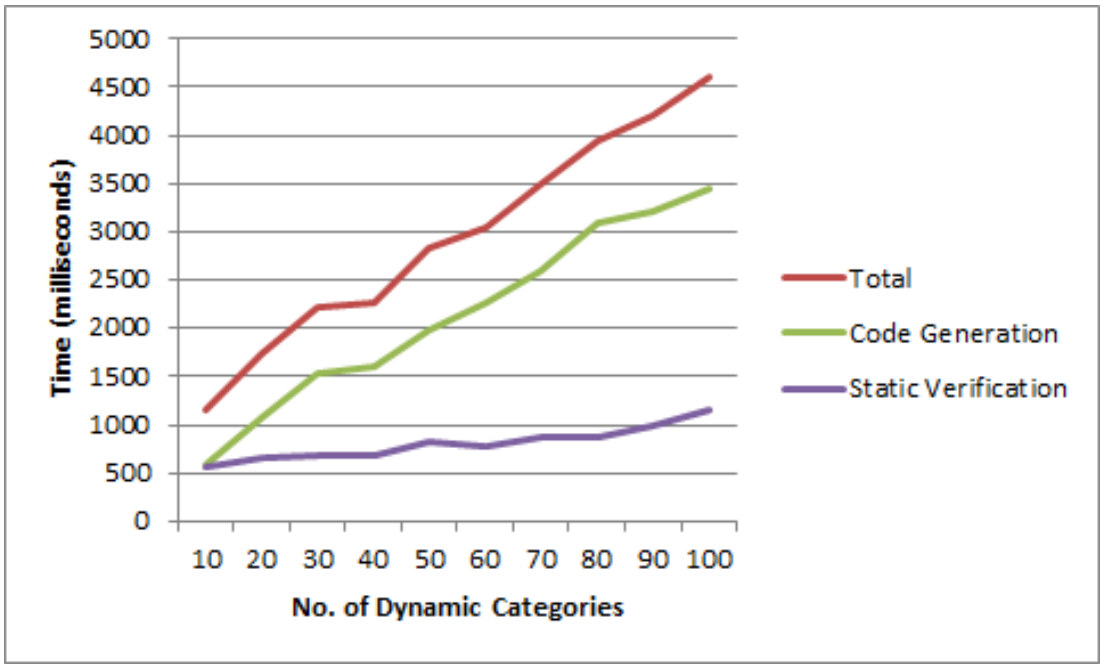

FIGURE 16.3: Runtime Performance Graph: Number of Dynamic Categories

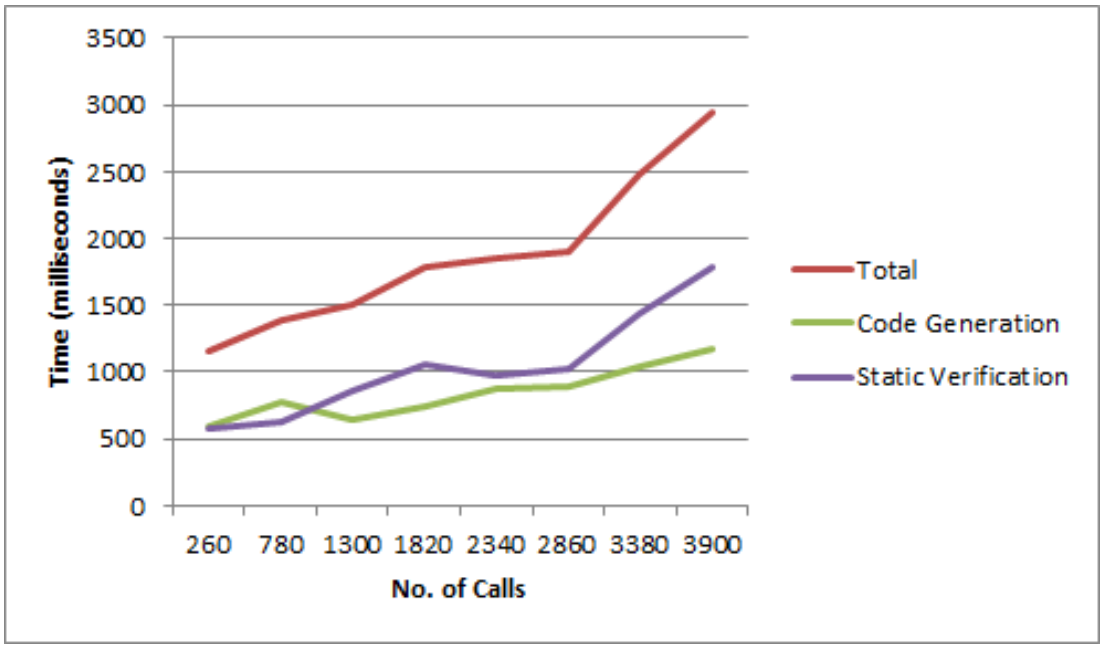

FIGURE 16.4: Runtime Performance Graph: Number of Calls

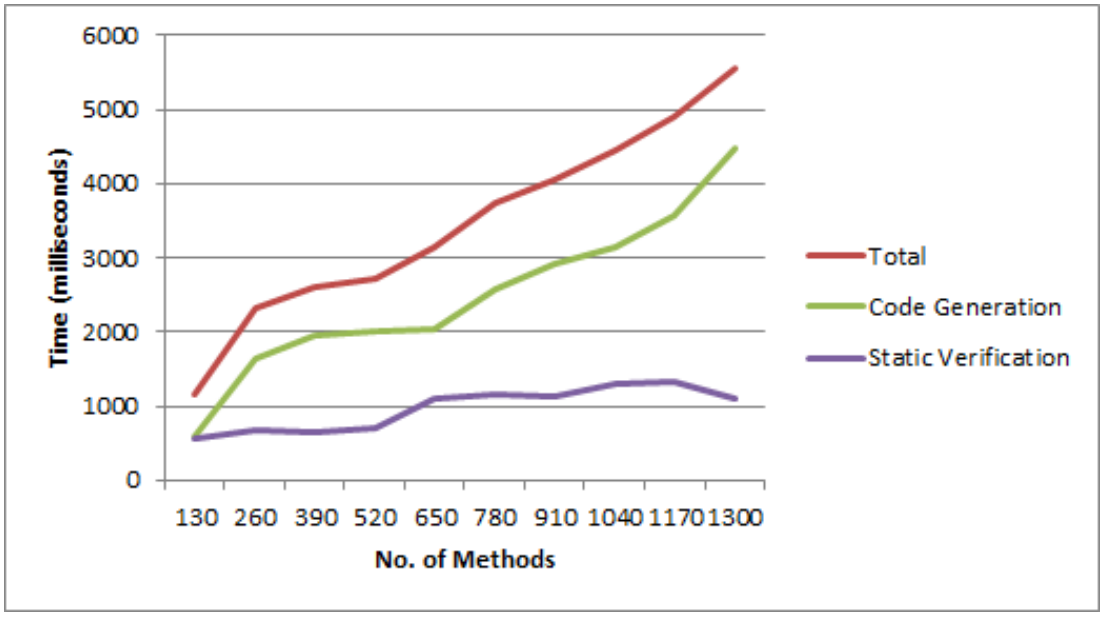

FIGURE 16.5: Runtime Performance Graph: Number of Methods

Note that tables containing the values for each of the tests are included in the appendix. By analysing the graphs, we can see that cases in which the the number of dynamic categories and the number of methods were increased had a bigger impact on the running time than the other cases. This is due to the fact that in these cases, there were more calls to actions in the program, therefore more points where code needed to be generated. It is clear that the generation of code is more expensive than the static checks. In these cases, the time taken for the static verifier checks also increased because there were more classes and method calls to check in the program. However, in the other cases, the time taken for the generation of code did not fluctuate significantly, because the number of points where code needed to be generated did not increase. In each case where there were increases in run-time performance (either of the static verification or the code generation), the time increased (fairly) linearly.

Part V

Conclusions and Future Work

Chapter 17

Conclusions

We have described two new systems. The first statically checks that a target program respects its RBAC policy. This system firstly consists of a policy language, JPol, in which a static RBAC policy can be expressed restricting the invocation of protected methods (actions) by roles.

Secondly, the target program must implement a set of patterns, which we name RBAC MVC, in order for it to be verified statically. These utilise the well known Model-View-Controller pattern but incorporate RBAC notions. They specify that there must be a class for each resource (which we called the resource class), and a model, controller and a set of views for each role (which we called the role model class, role controller class and role view classes). There is also a pattern for implementing the concept of users having multiple roles per session. These patterns help to incorporate RBAC security requirements into the design of the program and aid the process of statically verifying that the program respects the policy.

The last part of this first system is a static verification algorithm which we described at a high-level. This analyses the source code of the program to ensure it contains no

unauthorised invocations. Put simply, it checks that a set of classes belonging to one role can only invoke methods in each other (not classes belonging to other roles), or those actions in resource classes that are permitted for that role by the policy. Any classes that are not resource classes or do not belong to a role cannot invoke methods in these classes. Also, role classes can only be invoked in classes implementing the concept of session. These session classes form our trusted computing base - on which minimal checks are made.

If the program contains an unauthorised invocation to a resource class action or a method in a class belonging to a role, but not by a class belonging to that same role or a session class, then it will be rejected by the static verification algorithm. If the program successfully passes the static verifier's checks, then when using the program, the logged in user can only call those methods that have been authorised for the role currently activated for them, through the task methods in that role's model class. Therefore, no run-time access checks are needed. The static verifier catches policy violations early, at compile-time, aiding programmers in implementing policies correctly and nullifying run-time overheads attributed to dynamic enforcement mechanisms such as reference monitors.

The second system adapts and expands the previous system, which targets static policies and enforcement, to CBAC enforcement using a hybrid approach; combining compile-time and run-time enforcement mechanisms. This system also consists of a policy language, this time named JPolCat, in which a CBAC policy can be expressed restricting the invocation of protected methods (actions) by categories. Importantly, static and dynamic categories are declared separately. A static category is one to which every user assignment does not depend on conditions utilising dynamic information. A dynamic category is one to which at least one user is assigned depending on conditions utilising dynamic information. The policy also specifies the relationships between

categories, which arise due to CBAC. The first of these is hierarchical relationships, where one more senior static category 'subsumes' a subordinate static category or a more senior dynamic category 'subsumes' a subordinate static or dynamic category. The second is a *can-be* relation, where a user in a certain static or dynamic category can 'switch' to a dynamic category that is *can-be* related to the one they are in at that time. As with the static system, the policy only contains information needed for static checks, so it does not include user-to-category assignments or dynamic information. This is so that user-to-category assignments can change at run-time, allowing flexibility in this regard. We then establish an equivalence between static categories and roles, allowing us to re-use several concepts from the first, static system.

Secondly, the target program must implement a set of patterns, which we name CBAC MVC, in order for it to be verified using our hybrid mechanisms. These patterns are adapted and extended versions of the RBAC MVC patterns. These include, firstly, that the role MVC classes are adapted to categories, so that each category, both static and dynamic, is implemented into the program as a set of model, view and controller classes. Next, the relations between the categories are implemented as invocations in one category's MVC classes to the MVC classes of its related dynamic categories. Then, the session is implemented such that when users log-in, they can select any of their assigned static categories to interact with its set of MVC classes. From those, it can reach any of its related dynamic categories' MVC classes. The same applies from there, and so on. Lastly for the patterns, they specify patterns for a run-time security context and in-lined reference monitor - the latter of which we call the 'categoriser'. These patterns do not deviate significantly from previous patterns proposed in the literature for these concepts, but do have some restrictions to enable code to be generated easily into the program which invokes the categoriser. The categoriser is where the programmer implements the user-to-category assignments for dynamic categories (which includes the conditions for switching categories according to the category

relations discussed above).

Lastly for the second system is a static verification algorithm, which we also described at a high-level, which is also adapted from the static system. The checks are the same as before, except adapted to the category MVC classes and incorporating security context and categorises classes. In addition to the checks for the static case, the checks ensure that if a category class invokes a class belonging to a different category, the first category must be either hierarchically or *can-be* related to the category which the called class belongs to. No other classes can call resource classes or category MVC classes, except for session classes which may call the latter. This is so the session can direct the user to the MVC classes of a user-selected category. Also, all classes can call the security context classes, to update the session-scoped information and dynamic categories can call the categoriser, to perform the run-time checks. The static verification checks are followed by a code generation phase, where when a call to an action is found in a category MVC class belonging to a dynamic category, the entire method body in which the call is found is put into the true branch of an *if* statement. In this, the condition statement calls the categoriser to check if the user is a member of the category to which the class, in which the action is invoked, belongs. If true, the method body is executed and so the action is invoked. The else branch throws an exception, preventing the method body from being executed and so the action from being invoked. The static verification checks and code generation constitute the hybrid mechanism, which enforces static parts of a policy statically thus providing the benefits of the static system, but does not restrict expressivity of the policy language by allowing dynamic parts to be specified and then enforced dynamically. This also reduces the overall run-time impact of policy enforcement by reducing the amount of checks that can happen at run-time since all checks required for the static parts of the policy are done statically.

Chapter 18

Future Work

There are several ways we can improve, expand, adapt and apply the ideas in these systems in future work. Firstly, we can extend the static verifier with more checks within actions, to provide warnings to the programmer when there are indirect calls to actions within actions. It may be the case that one action a_1 calls another action a_2, but then a role/category may be permitted to invoke a_1 but **not** a_2. In this case, the programmer should be notified that a_1 calls a_2, so any role/category permitted to invoke a_1 should be permitted to invoke a_2 in the policy. Additionally, we can remove the reliance on access modifiers and extend the verifier to compensate for this. This will allow the approach to be usable in more languages i.e. those with no in-built access modifiers.

Secondly, we intend to formally specify the semantics of JPol/JPolCat and write the static verifier formally as a type checker. Thus, we can formally state the properties of the policy language, such as decidability and completeness, as well as the hybrid verification and formally prove those properties. Related to this, we can investigate utilising existing policy specification languages by developing translators to translate

the policy into JPolCat, or developing custom parsers to extract the data needed for our approach. In this way we can leverage the benefits of the existing languages, for example formally proved completeness and decidability. Coming back to the enforcement mechanism, we can present it using an aspect-oriented approach [18], allowing us to utilise existing aspect-oriented policy enforcement analysis tools and benefit from the performance gains this approach provides.

Thirdly, our approach has focused on the case of an application client, but we will investigate applying this approach to the web client case where a multitude of web technologies and languages can be used with JEE. We envisage that we will require separate parsers for each of the different possible languages that can be used with JEE.

Fourthly, our system will also be adapted to other platforms in addition to JEE, since the underlying concepts are independent of implementation platform (other than the requirements for an OO language with access modifiers).

In addition, we wish to apply our approach in specific domains, for example in real-world healthcare information systems, to test the approach in large-scale systems thus identifying and addressing the domain-specific challenges that will surely arise. For example, we can test how feasible the current target program patterns are to implement real-world large scale systems, improving on the shortcomings that may be found. Related to this, we can test the performance of a system that does not use our approach compared with the same system implemented with our approach, highlighting the performance gains.

We also plan to apply our approach to Java bytecode. This provides the significant benefit that our approach will apply to any language that compiles to bytecode and runs on the Java Virtual Machine - of which there are a multitude. This task provides some interesting challenges. The verifier would need to identify invocations of actions of resources in the bytecode. This is a significant challenge due to the difference in

structure between the Java source, which we target in our current approach, and the bytecode. Related to this, we can address the shortcoming attributed to Java's virtual method invocation. In this regard, we will benefit from the existing bytecode analysis tools [50], such as those that compute all the possible classes that a method invocation can be on. We can then incorporate this into the static checks.

Bibliography

[1] Tanvir Ahmed and Anand R. Tripathi. Static verification of security requirements in role based cscw systems. In *Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies*, SACMAT '03, pages 196–203, New York, NY, USA, 2003. ACM.

[2] Asad Ali and Maribel Fernández. Hybrid enforcement of category-based access control. In Sjouke Mauw and Christian Damsgaard Jensen, editors, *Security and Trust Management*, volume 8743 of *Lecture Notes in Computer Science*, pages 178–182. Springer International Publishing, 2014.

[3] Asad Ali and Maribel Fernández. Static enforcement of role-based access control. *arXiv preprint arXiv:1409.3533*, September 2014.

[4] Steve Barker. Action-status access control. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, pages 195–204, New York, NY, USA, 2007. ACM.

[5] Steve Barker. The next 700 access control models or a unifying meta-model? In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 187–196, New York, NY, USA, 2009. ACM.

[6] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, jan 2006.

[7] S. Berg. Ncsc-tg-004-88 glossary of computer security terms. Technical report, National Computer Security Center (NCSC), 1988.

[8] Elisa Bertino, Piero Andrea Bonatti, and Elena Ferrari. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, 4(3):191–233, August 2001.

[9] Clara Bertolissi and Maribel Fernández. A rewriting framework for the composition of access control policies. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '08, pages 217–225, New York, NY, USA, 2008. ACM.

[10] Clara Bertolissi and Maribel Fernández. Category-based authorisation models: Operational semantics and expressive power. In Fabio Massacci, Dan Wallach, and Nicola Zannone, editors, *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 140–156. Springer Berlin Heidelberg, 2010.

[11] Eric Bodden, Patrick Lam, and Laurie Hendren. Partially evaluating finite-state runtime monitors ahead of time. *ACM Trans. Program. Lang. Syst.*, 34(2):7:1–7:52, jun 2012.

[12] Piero A. Bonatti and Pierangela Samarati. Logics for authorizations and security. In Jan Chomicki, Ron van der Meyden, and Gunter Saake, editors, *Logics for Emerging Applications of Databases*, pages 277–323. Springer Berlin Heidelberg, 2004.

[13] Jean Bovet and Terence Parr. Antlrworks: An antlr grammar development environment. *Softw. Pract. Exper.*, 38(12):1305–1332, oct 2008.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[15] Robert Cartwright and Mike Fagan. Soft typing. *SIGPLAN Not.*, 39(4):412–428, April 2004.

[16] Maria Luisa Damiani, Elisa Bertino, Barbara Catania, and Paolo Perlasca. Geo-rbac: A spatially aware rbac. *ACM Trans. Inf. Syst. Secur.*, 10(1), February 2007.

[17] Sabrina De Capitani di Vimercati, Pierangela Samarati, and Sushil Jajodia. Policies, models, and languages for access control. In Subhash Bhalla, editor, *Databases in Networked Information Systems*, volume 3433 of *Lecture Notes in Computer Science*, pages 225–237. Springer Berlin Heidelberg, 2005.

[18] Anderson Santana de Oliveira, Eric Ke Wang, Claude Kirchner, and Helene Kirchner. Weaving rewrite-based access control policies. In *Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering*, FMSE '07, pages 71–80, New York, NY, USA, 2007. ACM.

[19] Eclipse Foundation. Eclipse - an open development platform. [Online]. Available: http://www.eclipse.org/ [Accessed: August. 18, 2014], 2014.

[20] Eduardo B. Fernandez, Tami Sorgente, and Maria M. Larrondo-Petrie. Even more patterns for secure operating systems. In *Proceedings of the 2006 Conference on Pattern Languages of Programs*, PLoP '06, pages 10:1–10:9, New York, NY, USA, 2006. ACM.

[21] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[22] David F. Ferraiolo, Ravi Sandhu, Serban Gavrila, D. Richard Kuhn, and Ramaswamy Chandramouli. Proposed nist standard for role-based access control. *ACM Trans. Inf. Syst. Secur.*, 4(3):224–274, aug 2001.

[23] Philip W.L. Fong. Relationship-based access control: Protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pages 191–202, New York, NY, USA, 2011. ACM.

[24] Luigi Giuri. Role-based access control in java. In *Proceedings of the Third ACM Workshop on Role-based Access Control*, RBAC '98, pages 91–100, New York, NY, USA, 1998. ACM.

[25] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley Java series. Addison-Wesley, 2003.

[26] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[27] Arun Gupta. *Java EE 7 Essentials.* O'Reilly Media, 2013.

[28] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, jan 2006.

[29] Vincent C Hu, David Ferraiolo, and D Richard Kuhn. *Assessment of access control systems.* US Department of Commerce, National Institute of Standards and Technology, 2006.

[30] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[31] JeeHyun Hwang, Tao Xie, Vincent Hu, and M. Altunay. Acpt: A tool for modeling and verifying access control policies. In *Policies for Distributed Systems and Networks (POLICY), 2010 IEEE International Symposium on*, pages 40–43, July 2010.

[32] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 2.* Springer-Verlag, London, UK, UK, 1995.

[33] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, aug 1988.

[34] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 426–441, London, UK, UK, 2002. Springer-Verlag.

[35] Makoto Murata, Akihiko Tozawa, Michiharu Kudo, and Satoshi Hada. Xml access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, August 2006.

[36] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 165–179, Washington, DC, USA, 2011. IEEE Computer Society.

[37] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[38] Marco Pistoia, Stephen J. Fink, Robert J. Flynn, and Eran Yahav. When role models have flaws: Static validation of enterprise security policies. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 478–488, Washington, DC, USA, 2007. IEEE Computer Society.

[39] Franois Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Programming Languages and Systems*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Berlin Heidelberg, 2001.

[40] Torsten Priebe, Eduardo B. Fernandez, Jens I. Mehlau, and Günther Pernul. A pattern system for access control. In *In Research Directions In Data And Applications Security XVIII, C. Farkas And P. Samarati (Eds.), Procs Of The 18th Annual IFIP WG 11.3 Working Conference On Data And Applications Security*, pages 25–28. Kluwer, 2004.

[41] Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD*, pages 137–196, 2000.

[42] A. Santana de Oliveira. Réécriture et Modularité pour les Politiques de Sécurité. PhD thesis, Université Henri Poincare, Nancy, France, 2008.

[43] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 86–101, London, UK, UK, 2001. Springer-Verlag.

[44] Basit Shafiq, Ammar Masood, James Joshi, and Arif Ghafoor. A role-based access control policy verification framework for real-time systems. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005), 2-4 February 2005, Sedona, AZ, USA*, pages 13–20, 2005.

[45] Igor Siveroni, Andrea Zisman, and George Spanoudakis. A uml-based static verification framework for security. *Requir. Eng.*, 15(1):95–118, 2010.

[46] Karsten Sohr, Michael Drouineaud, Gail-Joon Ahn, and Martin Gogolla. Analyzing and managing role-based access control policies. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):924–939, 2008.

[47] Christopher Steel, Ramesh Nagappan, and Ray Lai. *Core security patterns: Best practices and strategies for J2EE, Web services, and identity management.* Prentice Hall Core Series. Prentice-Hall, 2006.

[48] Rita C. Summers. *Secure Computing: Threats and Safeguards.* McGraw-Hill, Inc., Hightstown, NJ, USA, 1997.

[49] Fangqi Sun, Liang Xu, and Zhendong Su. Static detection of access control vulnerabilities in web applications. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 11–11, Berkeley, CA, USA, 2011. USENIX Association.

[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[51] Jeff Zarnett, Mahesh Tripunitara, and Patrick Lam. Role-based access control (rbac) in java via proxy objects using annotations. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, pages 79–88, New York, NY, USA, 2010. ACM.

Appendices

Appendix A

Tables of Runtime Performance

Session	Security Context	Categoriser	Others	Resources	SCategory	DCategory	Classes	Methods per Class	Methods Total	Calls per Method	Total Calls	Total Time (milli-seconds)	Code Generation	Static Verification
10	10	10	10	10	10	10	130	1	130	2	260	1163	591	572
10	10	10	10	20	10	10	140	1	140	2	280	1573	933	640
10	10	10	10	30	10	10	150	1	150	2	300	1711	682	1029
10	10	10	10	40	10	10	160	1	160	2	320	1681	781	900
10	10	10	10	50	10	10	170	1	170	2	340	1733	480	1253
10	10	10	10	60	10	10	180	1	180	2	360	1858	550	1308
10	10	10	10	70	10	10	190	1	190	2	380	2080	531	1549
10	10	10	10	80	10	10	200	1	200	2	400	2430	501	1929
10	10	10	10	90	10	10	210	1	210	2	420	2603	360	2243
10	10	10	10	100	10	10	220	1	220	2	440	3068	365	2703

FIGURE A.1: Runtime Performance Tests: Number of Resources

Session	Security Context	Categoriser	Others	Resources	SCategory	DCategory	Classes	Methods per Class	Methods Total	Calls per Method	Total Calls	Total Time (milli-seconds)	Code Generation	Static Verification
10	10	10	10	10	10	10	130	1	130	2	260	1163	591	572
10	10	10	10	10	20	10	160	1	160	2	320	1623	661	962
10	10	10	10	10	30	10	190	1	190	2	380	1647	622	1025
10	10	10	10	10	40	10	220	1	220	2	440	1667	654	1013
10	10	10	10	10	50	10	250	1	250	2	500	1398	514	884
10	10	10	10	10	60	10	280	1	280	2	560	1487	481	1006
10	10	10	10	10	70	10	310	1	310	2	620	1663	571	1092
10	10	10	10	10	80	10	340	1	340	2	680	1838	620	1218
10	10	10	10	10	90	10	370	1	370	2	740	2056	640	1416
10	10	10	10	10	100	10	400	1	400	2	800	1928	490	1438

FIGURE A.2: Runtime Performance Tests: Number of Static Categories

Session	Security Context	Categoriser	Others	Resources	SCategory	DCategory	Classes	Methods per Class	Methods Total	Calls per Method	Total Calls	Total Time (milli-seconds)	Code Generation	Static Verification
10	10	10	10	10	10	10	130	1	130	2	260	1163	591	572
10	10	10	10	10	10	20	160	1	160	2	320	1736	1075	661
10	10	10	10	10	10	30	190	1	190	2	380	2223	1532	691
10	10	10	10	10	10	40	220	1	220	2	440	2268	1593	675
10	10	10	10	10	10	50	250	1	250	2	500	2826	1992	834
10	10	10	10	10	10	60	280	1	280	2	560	3043	2273	770
10	10	10	10	10	10	70	310	1	310	2	620	3484	2603	881
10	10	10	10	10	10	80	340	1	340	2	680	3952	3081	871
10	10	10	10	10	10	90	370	1	370	2	740	4211	3217	994
10	10	10	10	10	10	100	400	1	400	2	800	4601	3449	1152

FIGURE A.3: Runtime Performance Tests: Number of Dynamic Categories

Session	Security Context	Categoriser	Others	Resources	SCategory	DCategory	Classes	Methods per Class	Methods Total	Calls per Method	Total Calls	Total Time (milli-seconds)	Code Generation	Static Verification
10	10	10	10	10	10	10	130	1	130	2	260	1163	591	572
10	10	10	10	10	10	10	130	1	130	6	780	1394	767	627
10	10	10	10	10	10	10	130	1	130	10	1300	1502	641	861
10	10	10	10	10	10	10	130	1	130	14	1820	1792	740	1052
10	10	10	10	10	10	10	130	1	130	18	2340	1846	871	975
10	10	10	10	10	10	10	130	1	130	22	2860	1908	888	1020
10	10	10	10	10	10	10	130	1	130	26	3380	2487	1045	1442
10	10	10	10	10	10	10	130	1	130	30	3900	2943	1164	1779

FIGURE A.4: Runtime Performance Tests: Number of Calls

Session	Security Context	Categoriser	Others	Resources	SCategory	DCategory	Classes	Methods per Class	Methods Total	Calls per Method	Total Calls	Total Time (milli-seconds)	Code Generation	Static Verification
10	10	10	10	10	10	10	130	1	130	2	260	1163	591	572
10	10	10	10	10	10	10	130	2	260	2	520	2321	1642	679
10	10	10	10	10	10	10	130	3	390	2	780	2598	1946	652
10	10	10	10	10	10	10	130	4	520	2	1040	2715	2003	712
10	10	10	10	10	10	10	130	5	650	2	1300	3139	2047	1092
10	10	10	10	10	10	10	130	6	780	2	1560	3733	2588	1145
10	10	10	10	10	10	10	130	7	910	2	1820	4061	2924	1137
10	10	10	10	10	10	10	130	8	1040	2	2080	4447	3156	1291
10	10	10	10	10	10	10	130	9	1170	2	2340	4904	3569	1335
10	10	10	10	10	10	10	130	10	1300	2	2600	5564	4471	1093

FIGURE A.5: Runtime Performance Tests: Number of Methods